A Gentle Introduction to Isabelle and Isabelle HOL

Gunnar Teege

March 24, 2024

Contents

Chapter 1

Isabelle System

Isabelle is a "proof assistant" for formal mathematical proofs. It supports a notation for propositions and their proofs, it can check whether a proof is correct, and it can even help to find a proof.

This introductory manual explains how to work with Isabelle to develop mathematical models. It does not presume prior knowledge about formal or informal proof techniques. It only assumes that the reader has a basic understanding of mathematical logics and the task of proving mathematical propositions.

1.1 Invoking Isabelle

After installation, Isabelle can be invoked interactively as an editor for entering propositions and proofs, or it can be invoked noninteractively to check a proof and generate a PDF document which displays the propositions and proofs.

1.1.1 Installation and Configuration

Isabelle is freely available from `https://isabelle.in.tum.de/` and other mirror sites for Windows, Mac, and Linux. It is actively maintained, there is usually one release every year. Older releases are available in a distribution archive.

To install Isabelle, follow the instructions on

```
https://isabelle.in.tum.de/installation.html
```

Although there are many configuration options, there is no need for an initial configuration, interactive and noninteractive invocation is immediately possible.

1.1.2 Theories and Sessions

The propositions and proofs in Isabelle notation are usually collected in “theory files” with names of the form `name.thy`. A theory file must import at least one other theory file to build upon its content. For theories based on higher order logic (“HOL”), the usual starting point to import is the theory *`Main`*.

Several theory files can be grouped in a “session”. A session is usually stored in a directory in the file system. It consists of a file named `ROOT` which contains a specification of the session, and the theory files which belong to the session.

When Isabelle loads a session it loads and checks all its theory files. Then it can generate a “heap file” for the session which contains the processed session content. The heap file can be reloaded by Isabelle to avoid the time and effort for processing and checking the theory files.

A session always has a single parent session, with the exception of the Isabelle builtin session `Pure`. Thus, every session depends on a linear sequence of ancestor sessions which begins at `Pure`. The ancestor sessions have separate heap files. A session is always loaded together with all ancestor sessions.

Every session has a name of the form `chap/sess` where `chap` is an arbitrary “chapter name”, it defaults to `Unsorted`. The session name and the name of the parent session are specified in the `ROOT` file in the session directory. When a session is loaded by Isabelle, its directory and the directories of all ancestor sessions must be known by Isabelle.

The Isabelle distribution provides heap files for the session `HOL/HOL` and its parent session `Pure/Pure`, the session directories are automatically known.

Every session may be displayed in a “session document”. This is a PDF document generated by translating the content of the session theory files to LaTeX. A frame LaTeX document must be provided which includes all content generated from the theory files. The path of the frame document, whether a session document shall be generated and which theories shall be included is specified in the `ROOT` file.

The command

```
isabelle mkroot [OPTIONS] [Directory]
```

can be used to initialize the given directory (default is the current directory) as session directory. It creates an initial `ROOT` file to be populated with theory file names and other specification for the session, and it creates a simple frame LaTeX document.

1.1.3 Invocation as Editor

Isabelle is invoked for editing using the command

```
  isabelle jedit [OPTIONS] [Files ...]
```

It starts an interactive editor and opens the specified theory files. If no file is specified it opens the file `Scratch.thy` in the user's home directory. If that file does not exist, it is created as an empty file.

The editor also loads a session (together with its ancestors), the default session to load is HOL. If a heap file exists for the loaded session it is used, otherwise a heap file is created by processing all the session's theories.

The default session to load can be changed by the option

```
  -l <session name>
```

Moreover the editor also loads (but does not open) theories which are transitively imported by the opened theory files. If these are Isabelle standard theories it finds them automatically. If they belong to the session in the current directory it also finds them. If they belong to other sessions, the option

```
  -d <directory pathname>
```

must be used to make the session directory known to Isabelle. For every used session a separate option must be specified.

If an imported theory belongs to the loaded session or an ancestor, it is directly referenced there. Otherwise the theory file is loaded and processed.

1.1.4 Invocation for Batch Processing

Isabelle is invoked for batch processing of all theory files in one or more sessions using the command

```
  isabelle build [OPTIONS] [Sessions ...]
```

It loads all theory files of the specified sessions and checks the contained proofs. It also loads all required ancestor sessions. If not known to Isabelle, the corresponding session directories must be specified using option `-d` as described in Section 1.1.3. Sessions required for other sessions are loaded from heap files if existent, otherwise the corresponding theories are loaded and a heap file is created.

If option `-b` is specified, heap files are also created for all sessions specified in the command. Option `-c` clears the specified sessions (removes their heap files) before processing them. Option `-n` omits the actual session processing, together with option `-c` it can be used to simply clear the heap files.

The specified sessions are only processed if at least one of their theory files has changed since the last processing or if the session is cleared using option

`-c`. If option `-v` is specified all loaded sessions and all processed theories are listed on standard output.

If specified for a session in its `ROOT` file (see Section 1.1.5), also the session document is generated when a session is processed.

1.1.5 Invocation for Document Creation

** todo **

1.2 Interactively Working with Isabelle

After invoking Isabelle as editor (see Section 1.1.3) it supports interactive work with theories.

The user interface consists of a text area which is surrounded by docking areas where additional panels can be displayed. Several panels can be displayed in the same docking area, using tabs to switch among them. Panels may also be displayed as separate undocked windows.

A panel can be displayed by selecting it in the `Plugins -> Isabelle` menu. Some of the panels are described in the following sections.

1.2.1 The Text Area

The text area displays the content of an open theory file and supports editing it. The font (size) used for display can be configured through the menu in `Utilities -> Global Options -> jEdit -> Text Area` together with many other options for display.

Processing the Text Area Content

Moreover, in the default configuration, Isabelle automatically processes the theory text up to the current part visible in the text area window. This includes processing the content of all imported theory files, if the import statement is visible.

Whenever the window is moved forward the processing is continued if "Continuous checking" has not been disabled in the Theories panel. Whenever the content of the text area is modified the processing is set back and restarted at the modified position.

In the default configuration the progress of the processing is shown by shading the unprocessed text in red and by a bar on the right border of the text area which symbolizes the whole theory file and shows the unprocessed part by red shading as well.

Displaying Definitions of Identifiers

Most identifiers used in Isabelle content have been defined in some theory file, this also holds for commands and other elements of the Isabelle syntax. The definition can be accessed by holding down CTRL (CMD on Macs) and clicking on an identifier. This also works for identifiers displayed in most other panels.

The definition is displayed by opening the corresponding theory file in the text area and positioning the window on the definition text.

If an identifier's definition has been loaded from a heap file (see Section 1.1.2) it is still displayed by opening the theory file, however, its content is not processed in the way described above. In particular, this is usually the case for all identifiers defined in the sessions `HOL/HOL` and `Pure/Pure`.

1.2.2 The Sidekick Panel

The Sidekick panel displays a structured view of the content of the text area and supports interactively expanding and collapsing substructures. It can be used for navigation in the text area by clicking on an item displayed in the Sidekick panel.

The structure view is not updated automatically upon changes in the the text area, to update it the text area content must be saved to its theory file.

1.2.3 The Output Panel

The Output panel displays the result of the theory text processing when it reaches the cursor position in the text area.

The displayed information depends on the cursor position and may be an information about the current theorem or proof or it may be an error message.

1.2.4 The State Panel

The State panel displays a specific result of the theory text processing if the cursor position is in a proof. It is called the "goal state" (see Section 2.2.1), and describes what remains to be proved by the rest of the proof.

The Output panel can be configured to include the goal state in its display by checking the "Proof state" button.

1.2.5 The Symbols Panel

Isabelle uses a large set of mathematical symbols and other special symbols which are usually not on the keyboard. The Symbols panel can be used to input such symbols in the text area. It comprises several tabs for selecting different symbol sets.

Alternatively, symbols not available on the keyboard may be entered by a specific sequence of keys, called an abbreviation. As an example, the sequence `==>` is an abbreviation for the symbol $\Longrightarrow$ consisting of three separate keys. In the interactive editor Isabelle replaces some abbreviations upon entering automatically by their symbol, others are left as they are.

If the mouse is positioned on a symbol in the Symbols panel available abbreviations are displayed as `abbrev:...` in the popup message.

1.2.6 The Documentation Panel

A comprehensive set of documentation about Isabelle can be opened through the Documentation panel. This manual refers to some of these documentations, if applicable.

For example, more information about the use of the interactive editor can be found in the Isabelle Reference Manual about jedit.

1.2.7 The Query Panel

The Query panel supports active searching for items in the content of all loaded sessions. There are several tabs for searching different kinds of items. Depending on the kind, a search specification must be entered in the tab, the results are displayed in the window below.

Note that a simple full text search is usually not supported. More information about supported search specifications for different kinds of items can be found in Chapter 2.

1.2.8 The Theories Panel

The Theories panel displays the loaded session and the opened or imported theories which do not belong to the loaded session or its ancestors. The (parts of) theories which have not been processed are shaded in red.

If the check button next to a theory is checked, the theory file is processed independently of the position of the text area window.

Chapter 2

Isabelle Basics

The basic mechanisms of Isabelle mainly support defining types, constants, and functions and specifying and proving statements about them.

2.1 Isabelle Theories

A theory is the content of an Isabelle theory file.

2.1.1 Theory Structure

The content of a theory file has the structure

```
theory name
imports name_1 ... name_n
begin
  ...
end
```

where `name` is the theory name and `name`$_1$ ... `name`$_n$ are the names of the imported theories. The theory name `name` must be the same which is used for the theory file, i.e., the file name must be `name.thy`.

The theory structure is a part of the Isabelle "outer syntax" which is mainly fixed and independent from the specific theories. Other kind of syntax is embedded into the outer syntax. The main embedded syntax is the "inner syntax" which is mainly used to denote types and terms. Content in inner syntax must always be surrounded by double quotes. The only exception is a single isolated identifier, for it the quotes may be omitted.

This introduction describes only a selected part of the outer syntax. The full outer syntax is described in the Isabelle/Isar Reference Manual and other documentation.

Additionally, text written in LaTeX syntax can be embedded into the outer syntax using the form **text**‹ ... › and LaTeX sections can be created using **chapter**‹ ... ›, **section**‹ ... ›, **subsection**‹ ... ›, **subsubsection**‹ ... ›, **paragraph**‹ ... ›, **subparagraph**‹ ... ›. Note that the delimiters used here are not the "lower" and "greater" symbols, but the "cartouche delimiters" available in the editor's Symbols panel in tab "Punctuation".

Embedded LaTeX text is intended for additional explanations of the formal theory content. It is displayed in the session document together with the formal theory content.

It is also possible to embed inner and outer syntax in the LaTeX syntax (see Chapter 4 in the Isabelle/Isar Reference Manual).

Moreover, comments of the form

```
(* ... *)
```

can be embedded into the outer syntax. They are only intended for the reader of the theory file and are not displayed in the session document.

Line breaks are ignored as part of the outer and inner syntax and have the same effect as a space.

2.1.2 Types

As usual in formal logics, the basic building blocks of propositions are terms. Terms denote arbitrary objects like numbers, sets, functions, or boolean values. Isabelle is strongly typed, so every term must have a type. However, in most situations Isabelle can derive the type of a term automatically, so that it needs not be specified explicitly. Terms and types are always denoted using the inner syntax.

Types are usually specified by type names. In Isabelle HOL (see Chapter 3) there are predefined types such as `nat` and `bool` for natural numbers and boolean values. New types can be defined in the form

typedecl *name*

which introduces the *name* for a new type for which the values are different from the values of all existing types and the set of values is not empty. No other information about the values is given, that must be done separately. See Chapter 4 for ways of defining types with specifying more information about their values.

Types can be parameterized, then the type arguments are denoted *before* the type name, such as in `nat set` which is the type of sets of natural numbers. A type name with *n* parameters is declared in the form

typedecl ('$name_1$,...,'$name_n$) *name* where the parentheses may be omitted if *n = 1*, such as in **typedecl** *'a set*. The type parameters are denoted by "type variables" which always have the form *'name* with a leading single quote character.

A type name with parameters is called a "type constructor" because it is not a type on its own. Every use where the parameters are replaced by actual types, such as in *nat set*, is called an "instance" of the parameterized type. If (some of) the parameters are replaced by type variables, such as in *'a set* or *('a set) set* or if a type is specified by a single type variable such as *'a* the type is called "polymorphic". A polymorphic type can be used as a type specification, its meaning is that an arbitrary instance can be used where the type variables are replaced by actual types.

Alternatively a type name can be introduced as a synonym for an existing type in the form

type_synonym *name = type*

such as in **type_synonym** *natset = nat set*. Type synonyms can also be parameterized as in

type_synonym *('name$_1$,...,'name$_n$) name = type*

where *type* may be a polymorphic type which contains atmost the type variables *'name$_1$,...,'name$_n$*.

2.1.3 Terms

Constants and Variables

Terms are mainly built as syntactical structures based on constants and variables. Constants are usually denoted by names, using the same namespace as type names. Whether a name denotes a constant or a type depends on its position in a term. Predefined constant names of type *bool* are *True* and *False*.

Constants of number types, such as *nat*, may be denoted by number literals, such as *6* or *42*.

A constant can be defined by specifying its type. The definition

consts *name$_1$ :: type$_1$ ... name$_n$ :: type$_n$*

introduces *n* constants with their names and types. No information is specified about the constant's values, in this respect the constants are "underspecified". The information about the values must be specified separately.

If the constant's type is polymorphic (see Section 2.1.2) the constant is also called polymorphic. Thus the declaration

consts *myset :: "'a set"*

declares the polymorphic constant `myset` which may be a set of elements of arbitrary type.

A (term) variable has the same form as a constant name, but it has not been introduced as a constant. Whenever a variable is used in a term it has a specific type which is either derived from its context or is explicitly specified in the form `varname :: type`.

Nested terms are generally written by using parentheses `(...)`. There are many priority rules how to nest terms automatically, but if in doubt, it is always safe to use parentheses.

Functions

A constant name denotes an object, which, according to its type, may also be a function of arbitrary order. Functions basically have a single argument. The type of a function is written as `argtype ⇒ restype`.

Functions in Isabelle are always total, i.e., they map every value of type `argtype` to some value of type `restype`. However, a function may be "underspecified" so that no information is (yet) available about the result value for some or all argument values. A function defined by

consts `mystery :: nat ⇒ nat`

is completely underspecified: although it maps every natural number to a unique other natural number no information about these numbers is available. Functions may also be partially specified by describing the result value only for some argument values. This does not mean that the function is "partial" and has no value for the remaining arguments. The information about these values may always be provided later, this does not "modify" the function, it only adds information about it.

Functions with Multiple Arguments

The result type of a function may again be a function type, then it may be applied to another argument. This is used to represent functions with more than one argument. Function types are right associative, thus a type $argtype_1 \Rightarrow argtype_2 \Rightarrow \cdots \Rightarrow argtype_n \Rightarrow restype$ describes functions which can be applied to n arguments.

Function application terms for a function `f` and an argument `a` are denoted by `f a`, no parentheses are required around the argument. Function application terms are left associative, thus a function application to n arguments is written $f\ a_1\ \ldots\ a_n$. Note that an application $f\ a_1\ \ldots\ a_m$ where $m < n$ (a "partial application") is a correct term and denotes a function taking the remaining `n-m` arguments.

For every constant alternative syntax forms may be defined for application terms. This is often used for binary functions to represent application terms in infix notation with an operator symbol. As an example, the name for the addition function is `plus`, so an application term is denoted in the form `plus 3 5`. For `plus` the alternative name `(+)` is defined (the parentheses are part of the name). For functions with such "operator names" an application term `(+) 3 5` can also be denoted in infix form `3 + 5`. Infix notation is supported for many basic functions and predicates, having operator names such as `(-)`, `(**)`, `(=)`, `(≠)`, `(≤)`, or `(∈)`.

Lambda-Terms

Functions can be denoted by lambda terms of the form `λx. term` where `x` is a variable which may occur in the `term`. The space between the dot and the `term` is often required to separate both. A function to be applied to `n` arguments can be denoted by the lambda term `λ`x_1 `...` x_n`. term` where x_1 ... x_n are distinct variables. As usual, types may be specified for (some of) the variables in the form `λ(`x_1`::`t_1`) ... (`x_n`::`t_n`). term`. The parentheses may be omitted if there is only one argument variable.

A constant function has a value which does not depend on the argument, thus the variable `x` does not occur in the `term`. Then its name is irrelevant and it may be replaced by the "wildcard" `_` (an underscore) as in `λ_. term`.

Searching Constants

Constants may be searched using the command

find_consts `criterion`$_1$ `...` `criterion`$_n$

or using the Query panel (see Section 1.2.7) in the "Find Constants" tab using the sequence `criterion`$_1$ `...` `criterion`$_n$ as search specification in the "Find:" input field. The `criterion`$_i$ are combined by conjunction.

The command **find_consts** may be entered in the text area between other theory content such as type or constant declarations. It finds all named constants which have been introduced before the command position. Searches using the Query panel find all named constants which have been introduced before the cursor position in the text area.

A `criterion`$_i$ may be a type, specified in inner syntax and quoted if not a single type name. Then the search finds all constants where the type occurs as a part of the constant's type. For example, it finds all functions which have the specified type as argument or result type.

A `criterion`$_i$ may also have the form `strict: "type"`, then the search only finds constants which have that type. In both cases the specified type may be a function type, then the search finds corresponding named functions.

If the specified type is polymorphic the search will also find constants which have an instance of it as their type or as a part of the type, respectively.

A *criterion*$_i$ may also have the form *name: strpat* where *strpat* is a string pattern which may use "*" as wildcard (then the pattern must be enclosed in double quotes). Then all constants are found where the *strpat* matches a substring of their name.

2.1.4 Definitions and Abbreviations

A constant name may be introduced together with information about its associated value by specifying a term for the value. There are two forms for introducing constant names in this way, definitions and abbreviations.

Definitions

A definition defines a new constant together with its type and value. It is denoted in the form

definition *name :: type*
where *"name* $\equiv$ *term"*

Note that the "defining equation" *name* $\equiv$ *term* is specified in inner syntax and, like *type* must be delimited by quotes. The *name* may not occur in the *term*, i.e., this form of definition does not support recursion.

If the type of the defined name is a function type, the *term* may be a lambda term. Alternatively, the definition for a function applicable to *n* arguments can be written in the form

definition *name :: type*
where *"name* $x_1 \ldots x_n \equiv$ *term"*

with variable names $x_1 \ldots x_n$ which may occur in the *term*. This form is mainly equivalent to

definition *name :: type*
where *"name* $\equiv \lambda x_1 \ldots x_n.$ *term"*

A short form of a definition is

definition *"name* $\equiv$ *term"*

Here, the type of the new constant is derived as the type of the *term*.

Usually, a constant defined in this way is fully specified, i.e., all information about its value is available. However, if the term does not provide this information, the constant is still underspecified. Consider the definition

definition *"mystery2 ≡ mystery"*

where `mystery` is defined as above. Then it is only known that `mystery2` has type `nat ⇒ nat` and is the same total function as `mystery`, but nothing is known about its values.

Abbreviations

An abbreviation definition does not define a constant, it only introduces the name as a synonym for a term. Upon input the name is automatically expanded, and upon output it is used whenever a term matches its specification and the term is not too complex. An abbreviation definition is denoted in a similar form as a definition:

abbreviation *name :: type*
where *"name ≡ term"*

As for definitions, recursion is not supported, the `name` may not occur in the `term`. The short form is also available as for definitions.

The alternative form for functions is also available. The abbreviation definition

abbreviation *name :: type*
where *"name* x_1 ... x_n *≡ term"*

introduces a “parameterized” abbreviation. An application term *name* $term_1$ ... $term_n$ is replaced upon input by `term` where all occurrences of x_i have been substituted by $term_i$. Upon output terms are matched with the structure of `term` and if successful a corresponding application term is constructed and displayed.

2.1.5 Overloading

True Overloading

One way of providing information about the value of an underspecified constant is overloading. It provides the information with the help of another constant together with a definition for it.

Overloading depends on the type. Therefore, if a constant is polymorphic, different definitions can be associated for different type instances.

Overloading is only possible for constants which do not yet have a definition, i.e., they must have been defined by **consts** (see Section 2.1.3). Such a constant `name` is associated with *n* definitions by the following overloading specification:

```
overloading
  name₁ ≡ name
    ...
  nameₙ ≡ name
begin
  definition name₁ :: type₁ where ...
    ...
  definition nameₙ :: typeₙ where ...
end
```

where all `type`$_i$ must be instances of the type declared for `name`.

The auxiliary constants `name`$_1$ ... `name`$_n$ are only introduced locally and cannot be used outside of the **overloading** specification.

Adhoc Overloading

There is also a form of overloading which achieves similar effects although it is implemented completely differently. It is only performed on the syntactic level, like abbreviations. To use it, the theory *HOL-Library.Adhoc_Overloading* must be imported by the surrounding theory:

imports *"HOL-Library.Adhoc_Overloading"*

(Here the theory name must be quoted because it contains a minus sign.)

Then a constant name `name` can be defined to be a "type dependent abbreviation" for `n` terms of different type instances by

adhoc_overloading *name term*$_1$ ... *term*$_n$

Upon input the type of `name` is determined from the context, then it is replaced by the corresponding `term`$_i$. Upon output terms are matched with the corresponding `term`$_i$ and if successful `name` is displayed instead.

Although `name` must be the name of an existing constant, only its type is used. The constant is not affected by the adhoc overloading, however, it becomes inaccessible because its name is now used as term abbreviation.

Several constant names can be overloaded in a common specification:

adhoc_overloading *name*$_1$ *term*$_{11}$ ... *term*$_{1n}$ **and** ... **and** *name*$_k$ ...

2.1.6 Propositions

A proposition denotes an assertion, which can be valid or not. Valid proposition are called "facts", they are the main content of a theory. Propositions are specific terms and are hence written in inner syntax and must be enclosed in quotes.

Formulas

In its simplest form a proposition is a single term of type `bool`, such as

`6 * 7 = 42`

Terms of type `bool` are also called “formulas”.

A proposition may contain free variables as in

`2 * x = x + x`

A formula as proposition is valid if it evaluates to `True` for all possible values substituted for the free variables.

Derivation Rules

More complex propositions can express, “derivation rules” used to derive propositions from other propositions. Derivation rules are denoted using a “metalogic language”. It is still written in inner syntax but uses a small set of “metalogic operators”.

Derivation rules consist of assumptions and a conclusion. They can be written using the metalogic operator $\Longrightarrow$ in the form

$A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow C$

where the A_1 … A_n are the assumptions and C is the conclusion. The conclusion must be a formula. The assumptions may be arbitrary propositions. If an assumption contains metalogic operators parentheses can be used to delimit them from the rest of the derivation rule.

A derivation rule states that if the assumptions are valid, the conclusion can be derived as also being valid. So it can be viewed as a “meta implication” with a similar meaning as a boolean implication, but with a different use.

An example for a rule with a single assumption is

$(x::nat) < c \Longrightarrow n*x \leq n*c$

Note that type `nat` is explicitly specified for variable `x`. This is necessary, because the constants `<`, `*`, and $\leq$ are overloaded and can be applied to other types than only natural numbers. Therefore the type of `x` cannot be derived automatically. However, when the type of `x` is known, the types of `c` and `n` can be derived to also be `nat`.

An example for a rule with two assumptions is

$(x::nat) < c \Longrightarrow n > 0 \Longrightarrow n*x < n*c$

In most cases the assumptions are also formulae, as in the example. However, they may also be again derivation rules. Then the rule is a “meta rule” which derives a proposition from other rules.

Binding Free Variables

A proposition may contain universally bound variables, using the metalogic quantifier $\bigwedge$ in the form

$\bigwedge x_1 \ldots x_n .\ P$

where the $x_1 \ldots x_n$ may occur free in the proposition P. As usual, types may be specified for (some of) the variables in the form $\bigwedge (x_1::t_1) \ldots (x_n::t_n).\ P$. An example for a valid derivation rule with bound variables is

$\bigwedge$ (x::nat) c n . x < c $\Longrightarrow$ n*x $\le$ n*c

If a standalone proposition in a theory contains free variables they are implicitly universally bound. Thus the example derivation rule above is equivalent to the single-assumption example rule in the previous section. Explicit binding of variables is only required to avoid name clashes with constants of the same name. In the proposition

$\bigwedge$ (True::nat). True < c $\Longrightarrow$ n*True $\le$ n*c

the name `True` is used locally as a variable of type `nat` instead of the predefined constant of type `bool`. Of course, using well known constant names as variables is confusing and should be avoided.

Alternative Rule Syntax

An alternative, Isabelle specific syntax for derivation rules is

$\bigwedge x_1 \ldots x_m .\ [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow C$

which is often considered as more readable, because it better separates the assumptions from the conclusion. In the interactive editor to switch to this form it may be necessary to set `Print Mode` to `brackets` in `Plugin Options` for `Isabelle General`. The fat brackets are available for input in the editor's Symbols panel in tab “Punctuation”.
Using this syntax the two-assumption example rule from the previous section is denoted by

$\bigwedge$ (x::nat) c n. $[\![$x < c; n > 0$]\!]$ $\Longrightarrow$ n*x < n*c

or equivalently without quantifier by

$[\![$(x::nat) < c; n > 0$]\!]$ $\Longrightarrow$ n*x < n*c

Note that in the literature a derivation rule $[\![P;\ Q]\!] \Longrightarrow P \wedge Q$ is often denoted in the form

$$\frac{P \qquad Q}{P \wedge Q}$$

Structured Rule Syntax

Isabelle supports another alternative syntax for derivation rules. It is called “structured” form, since the rule is not specified by a single proposition but by several separate propositions for the parts of the rule:

"C" **if** "A_1" ... "A_n" **for** x_1 ... x_m

Here the assumptions and the variables may be grouped or separated for better readability by the keyword **and**. For every group of variables (but not for single variables in a group) a type may be specified in the form x_1 ... x_m `:: "type"`, it applies to all variables in the group.

The keywords **if**, **and**, **for** belong to the outer syntax. Thus, a rule in structured form cannot occur nested in another proposition, such as an assumption in another rule. Moreover, the original rule must be quoted as a whole, whereas in the structured form only the sub-propositions C, A_1, ..., A_n must be individually quoted. The x_1, ..., x_m need not be quoted, but if a type is specified for a variable group the type must be quoted, if it is not a single type name.

If written in this form, the two-assumption example rule from the previous section may become

`"n*x < n*c"` **if** `"x < c"` **and** `"n > 0"` **for** `x::nat` **and** `n c`

The assumptions and the conclusion in a rule in structured form may be arbitrary propositions, in particular, they may be derivation rules (in unstructured form). If the conclusion is a derivation rule its assumptions are prepended to the assumptions which are already present in the rule.

2.1.7 Theorems

A theorem specifies a proposition together with a proof, that the proposition is valid. Thus it adds a fact to the enclosing theory. A simple form of a theorem is

theorem `"prop"` ⟨*proof*⟩

where `prop` is a proposition in inner syntax and ⟨*proof*⟩ is a proof as described in Section 2.2. The keyword **theorem** can be replaced by one of the keywords **lemma**, **corollary**, **proposition** to give a hint about the use of the statement to the reader.

If the proposition in a theorem is a derivation rule it may also be specified in structured form (see Section 2.1.6):

theorem "C" **if** "A_1" ... "A_n" **for** x_1 ... x_m ⟨*proof*⟩

Unknowns

Whenever a theorem turns a proposition to a fact, the free (or universally bound) variables are replaced by "unknowns". For a variable `name` the corresponding unknown is `?name`. This is only a technical difference, it signals to Isabelle that the unknowns can be consistently substituted by arbitrary terms, as long as the types are preserved.

The result of such a substitution is always a special case of the fact and therefore also a fact. In this way a fact with unknowns gives rise to a (usually infinite) number of facts which are constructed by substituting unknowns by terms.

When turned to a fact, the example rule from the previous sections becomes

```
?x < ?c ⟹ ?n*?x ≤ ?n*?c
```

with type `nat` associated to all unknowns.

Propositions specified in a theorem may not contain unknowns, they are only introduced by Isabelle after proving the proposition.

Isabelle can be configured to suppress the question mark when displaying unknowns, then this technical difference becomes invisible.

Named Facts

Facts are often used in proofs of other facts. For this purpose they can be named so that they can be referenced by name. A named fact is specified by a theorem of the form

theorem *name: "prop"* ⟨*proof*⟩

The names used for facts have the same form as names for constants and variables (see Section 2.1.3). The same name can be used for a variable and a fact, they can always be distinguished by the usage context.

The example rule from the previous sections can be turned into a fact named *example1* by

theorem *example1: "(x::nat) < c ⟹ n*x ≤ n*c"* ⟨*proof*⟩

It is also possible to introduce named collections of facts. A simple way to introduce such a named collection is

lemmas *name* = *name*$_1$ ... *name*$_n$

where *name*$_1$ ... *name*$_n$ are names of existing facts or fact collections.

If there is a second rule stated as a named fact by

theorem *example2: "(x::nat) ≤ c ⟹ x + m ≤ c + m"* ⟨*proof*⟩

a named collection can be introduced by

lemmas *examples = example1 example2*

Alternatively a "dynamic fact set" can be declared by

named_theorems *name*

It can be used as a "bucket" where facts can be added afterwards by specifying the bucket name in the theorem:

theorem *[name]: "prop"* ⟨*proof*⟩

or together with specifying a fixed fact name $name_f$ by

theorem $name_f$ *[name]: "prop"* ⟨*proof*⟩

There are also some predefined "internal fact sets". For them the name can only be used to add facts as described above, the set cannot be used or displayed by referring it by name. Examples are the internal fact sets *intro* (see Section 2.3.3) and *simp* (see Section 2.3.6).

Alternative Theorem Syntax

If the proposition of a theorem is a derivation rule Isabelle supports an alternative structured form for it:

theorem
  fixes x_1 ... x_m
  assumes "A_1" ... "A_n"
  shows "*C*"
  ⟨*proof*⟩

Like for the general structured form (see Section 2.1.6) the variables and assumptions may be grouped by **and**, a type may be specified for each variable group, the keywords belong to the outer syntax and the C, A_1, ..., A_n must be individually quoted. Note that this structured form may only be used if a proposition is specified in a theorem.

Using this syntax the two-assumption example rule from Section 2.1.6 can be written as

theorem
  fixes *x::nat* **and** *c n*
  assumes *"x < c"* **and** *"n > 0"*
  shows *"n*x < n*c"*
  ⟨*proof*⟩

Note that a name specified for the conclusion as

```
shows name: "C"
```

becomes the name for the whole fact introduced by the theorem, not only for the conclusion. Alternatively the name for the fact can be specified in the usual way after the **theorem** keyword:

```
theorem name:
  fixes x₁ ... xₘ
  assumes "A₁" ... "Aₙ"
  shows "C"
  ⟨proof⟩
```

Definitions as Facts

The definitions described in Section 2.1.4 also introduce facts in the enclosing theory. Every definition introduces a new constant and specifies a defining equation of the form *name* ≡ *term* for it. This equation is a proposition using the "meta equality" ≡ which is another metalogic operator. It is the initial information given for the new constant, thus it is valid "by definition" and is a fact in the theory.

These facts are automatically named. If *name* is the name of the defined constant, the defining equation is named *name_def*. Alternatively an explicit name can be specified in the form

```
definition name :: type
where fact_name: "name ≡ term"
```

Although the auxiliary constants used in an **overloading** specification (see Section 2.1.5) are not accessible outside the specification, their definitions are. So they can be referred by their names and used as information about the overloaded constant.

Displaying and Searching Named Facts

A named fact or fact set (but not a dynamic fact set) can be displayed in its standard form as proposition using the command

```
thm name
```

and it can be displayed in its structured form with **fixes**, **assumes**, and **shows** using the command

```
print_statement name
```

Named facts may be searched using the command

find_theorems *`criterion`*$_1$ ... *`criterion`*$_n$

or using the Query panel (see Section 1.2.7) in the “Find Theorems” tab using the sequence *`criterion`*$_1$ ... *`criterion`*$_n$ as search specification in the “Find:” input field. The *`criterion`*$_i$ are combined by conjunction.

A *`criterion`*$_i$ may be a term containing unknowns as subterms (called a “term pattern”). Then all facts are found which contain a matching subterm in their proposition. A term pattern matches a subterm if the unknowns in the pattern can be consistently replaced by terms so that the result is syntactically equal to the subterm. The term pattern is specified in inner syntax and must be quoted. Only named facts can be found in this way.

The example theorems *`example1`* and *`example2`* can be found using the term pattern *`"?t1 ≤ ?t2"`*, whereas the pattern *`"?t1 + ?c ≤ ?t2 + ?c"`* will only find *`example2`*.

A *`criterion`*$_i$ may also have the form *`name: strpat`* where *`strpat`* is a string pattern which may use “*” as wildcard (then the pattern must be enclosed in double quotes). Then all facts are found where the *`strpat`* matches a substring of the fact name. After naming the example theorems as above the criterion *`name: example`* will display the theorems *`example1`* and *`example2`* with their names and propositions.

The commands for display and search may be entered in the text area outside of theorems and at most positions in a proof. The found facts are displayed with their names in the Output panel (see Section 1.2.3).

2.2 Isabelle Proofs

Every proposition stated as a fact in an Isabelle theory must be proved immediately by specifying a proof for it. A proof may have a complex structure of several steps and nested subproofs, its structure is part of the outer syntax.

2.2.1 Maintaining Proof State

Usually it is necessary during a proof to collect information for later use in the proof. For every proof such state is maintained in two structures: the “proof context” and the “goal state”. At the end of a proof all proof state is disposed, only the proved fact remains in the enclosing environment.

Proof Context

The proof context is similar to a temporary theory which collects facts and other proof elements. It may contain

- Facts: as usual, facts are valid propositions. However, they need not be globally valid, they can be assumed to be only valid locally during the proof. Like in a theory facts and fact sets may be named in a proof context.
- Fixed variables: fixed variables are used to denote the "arbitrary but fixed" objects often used in a proof. They can be used in all facts in the same proof context. They can be roughly compared to the constants in a theory.
- Term abbreviations: these are names introduced locally for terms. They can be roughly compared to abbreviations defined in a theory. Using such names for terms occurring in propositions it is often possible to denote propositions in a more concise form.

Like in a theory the names for facts and fixed variables have the same form, they can always be distinguished by their usage context. The names for term abbreviations have the form of unknowns (see Section 2.1.7) and are thus always different from variable names.

Since the proof context is usually populated by explicitly specifying its elements it is visible in the proof text and also in the session document. In the interactive editor a list of all elements of the proof context at the cursor position can be obtained in the Query panel (see Section 1.2.7) in tab "Print Context" by checking "terms" (for term abbreviations) and/or "theorems" (for facts).

Goal State

The goal state is used to collect propositions which have not yet been proved. It is used in the form of a "to-do list". It is the duty of a proof to prove all goals in its goal state. During the proof goals may be removed from the goal state or may be added. A proof may be terminated when its goal state is empty.

The content of the goal state is not maintained by explicit specifications of the proof writer, it is updated implicitly by the Isabelle proof mechanism. As a consequence it is usually not visible in the session document. In the interactive editor it is displayed in the State panel (see Section 1.2.4) or in the Output panel (see Section 1.2.3).

If a subproof is nested in another proof the goal state of the inner proof hides the goal state of the outer proof until the inner proof is complete.

Initial Proof State

The initial proof state in a theorem of the form

theorem "$\bigwedge$ x_1 ... x_m. $[\![A_1;$ ...; $A_n]\!]$ $\Longrightarrow$ C" ⟨*proof*⟩

has the proposition $\bigwedge$ x_1 ... x_m. $[\![A_1;$ ...; $A_n]\!]$ $\Longrightarrow$ C as the only goal in the goal state and an empty proof context.

If the proposition of a theorem is specified in structured form

theorem "C" **if** "A_1" ... "A_n" **for** x_1 ... x_m ⟨*proof*⟩

or

theorem
  fixes x_1 ... x_m **assumes** "A_1" ... "A_n" **shows** "C" ⟨*proof*⟩

the initial goal state only contains the conclusion C, whereas the initial proof context contains the assumptions A_1, ..., A_n as (assumed) facts and the variables x_1 ... x_m as fixed variables.

Both structured forms support naming the assumptions in the proof context. Every assumption group separated by **and** may be given a name, i.e., the assumptions may be specified in the form

if $name_1$: "A_{11}" ... "A_{1m1}" **and** ... **and** $name_n$: "A_{n1}" ... "A_{nmn}"

or

assumes $name_1$: "A_{11}" ... "A_{1m1}" **and** ... **and** $name_n$: "A_{n1}" ... "A_{nmn}"

respectively.

Additionally, Isabelle always automatically names the assumptions in all groups together. For the structured form beginning with **if** it uses the name *that*, for the structured form beginning with **assumes** it uses the name *assms*.

Since the assumption names are only defined in the proof context they can only be used locally in the proof of the theorem. Therefore, if the general structured form of a proposition beginning with **if** is used in a context where no proof is required, such as in an **assume** statement (see Section 2.2.8), it is not possible to specify names for the assumption groups.

2.2.2 Proof Procedure

Assume you want to prove a derivation rule A $\Longrightarrow$ C with a single assumption A and the conclusion C. The basic procedure to build a proof for it is to construct a sequence of the form $F_1 \Longrightarrow F_2$, $F_2 \Longrightarrow F_3$, $F_3 \Longrightarrow \cdots \Longrightarrow F_{n-1}$, $F_{n-1} \Longrightarrow F_n$ from rules $RA_i \Longrightarrow RC_i$ for i=1...n-1 which are already known to be valid (i.e., facts) where F_1 matches with A and RA_1, F_n matches with C and RC_{n-1}, and every other F_i matches with RA_i and RC_{i-1}.

The sequence can be constructed from left to right (called "forward reasoning") or from right to left (called "backward reasoning") or by a combination of both.

Consider the rule `(x::nat) < 5` $\Longrightarrow$ `2*x+3` $\leq$ `2*5+3`. A proof can be constructed from the two example rules `example1` and `example2` from the previous sections as the sequence `(x::nat) < 5` $\Longrightarrow$ `2*x` $\leq$ `2*5`, `2*x` $\leq$ `2*5` $\Longrightarrow$ `2*x+3` $\leq$ `2*5+3` consisting of three facts.

Forward reasoning starts by assuming `A` to be the local fact F_1 and incrementally constructs the sequence from it. An intermediate result is a part `A`, F_2, ..., F_i of the sequence, here F_i is the "current fact". A forward reasoning step consists of stating a proposition F_{i+1} and proving it to be a new local fact from the current fact F_i using a valid rule $RA_i \Longrightarrow RC_i$. The step results in the extended sequence `A`, F_2, ..., F_i, F_{i+1} with new current fact F_{i+1}. When a step successfully proves a current fact F_n which matches the conclusion `C` the proof is complete.

Backward reasoning starts at the conclusion `C` and incrementally constructs the sequence from it backwards. An intermediate result is a part F_i, ..., F_{n-1}, `C` of the sequence, here F_i is the "current goal". A backward reasoning step consists of constructing a new current goal F_{i-1} and the extended sequence F_{i-1}, F_i, ..., F_{n-1}, `C` using a valid rule $RA_{i-1} \Longrightarrow RC_{i-1}$. When a step produces a new current goal F_1, which matches the assumption `A`, the proof is complete.

Unification

The matching at the beginning and end of the sequence and when joining the used rules is done by "unification". Two propositions `P` and `Q` are unified by substituting terms for unknowns in `P` and `Q` so that the results become syntactically equal.

Since only the $RA_i \Longrightarrow RC_i$ are facts containing unknowns, only they are modified by the unification, `A` and `C` remain unchanged.

Note that when an unknown is substituted by a term in RA_i, the same unknown must be substituted by the same term in RC_i and vice versa, to preserve the validness of the rule $RA_i \Longrightarrow RC_i$. In other words, the sequence is usually constructed from specializations of the facts $RA_i \Longrightarrow RC_i$ where every conclusion is syntactically equal to the assumption of the next rule.

In the example the assumption `?x < ?c` of rule `example1` is unified with `(x::nat) < 5` by substituting the term `5` for the unknown `?c`, and the variable `x` for the unknown `?x` resulting in the specialized rule `(x::nat) < 5` $\Longrightarrow$ `n*x` $\leq$ `n*5`. The conclusion `?x + ?m` $\leq$ `?c + ?m` of rule `example2` is unified with `2*x+3` $\leq$ `2*5+3` by substituting the term `2*x` for the unknown `?x`, the term `2*5` for the unknown `?c`, and the term `3` for the unknown `?m` resulting in the specialized rule `2*x` $\leq$ `2*5` $\Longrightarrow$ `2*x+3` $\leq$ `2*5+3`. Now the two specialized rules can be joined by substituting the term 2 for the unknown `?n` in the first, resulting in the sequence which constitutes the proof.

Storing Facts During a Proof

In a proof for a derivation rule $A \Longrightarrow C$ the assumption A, the conclusion C and the intermediate facts $F_1, F_2, \ldots, F_n$ constructed by the proof steps must be stored. There are mainly two ways how this can be done in an Isabelle proof.

The first way is to store the facts at the beginning of the sequence in the proof context and the facts at the end of the sequence in the goal state. Initially, A is the only fact in the proof context and C is the only goal in the goal state. A forward reasoning step consists of adding fact F_{i+1} to the proof context and proving its validness using rule $RA_i \Longrightarrow RC_i$. The goal state remains unchanged. A backward reasoning step consists of replacing the current goal F_i by the new current goal F_{i-1} in the goal state and proving that the new goal implies the previous one using rule $RA_{i-1} \Longrightarrow RC_{i-1}$. The proof context remains unchanged. The proof is complete when the current goal matches (unifies with) a fact in the proof context.

Note that the facts in the proof context and in the goal state are treated differently. A backward step replaces the goal since the old goal needs not be handled again in the proof. Whenever the new goal has been proved the old goal is known to be valid as well. Since the goal state is used to determine when the proof is complete, it is crucial to remove all unnecessary goals from it. A forward step, instead, adds a fact to the proof context. It could remove the previous facts, since they are not needed in the special case described here, however, there are proofs where facts are used more than once, therefore it is usually useful to keep them in the proof context, and it is irrelevant for detecting whether a proof is complete.

The second way is to store all facts in the goal state by using a current goal of the form $[\![F_1; \ldots; F_i]\!] \Longrightarrow F_{i+j}$, i.e., a derivation rule. The proof context is not used at all. Initially, the goal state contains the goal $A \Longrightarrow C$. A forward reasoning step consists of adding fact F_{i+1} as assumption to the current goal and proving its validness as above. A backward reasoning step consists of replacing the conclusion F_{i+j} of the current goal by F_{i+j-1} and proving that it implies F_{i+j} as above. The proof is complete when the conclusion of the current goal unifies with one of its assumptions.

Note that these two ways correspond to the initial proof states prepared by the different forms of theorems. The basic form **theorem** "$A \Longrightarrow C$" puts $A \Longrightarrow C$ into the goal state and leaves the proof context empty, as required for the second way. The structured forms, such as **theorem** "C" **if** "A" put A into the proof context and C into the goal state, as required for the first way.

Multiple Assumptions

If the rule to be proved has more than one assumption `A` the sequence to be constructed becomes a tree where the branches start at (copies of) the assumptions $A_1, \ldots, A_n$ and merge to finally lead to the conclusion `C`. Two branches which end in facts F_{1n} and F_{2m} are joined by a step $\llbracket F_{1n}; F_{2m} \rrbracket \Longrightarrow F_1$ to a common branch which continues from fact F_1.

A proof which constructs this tree may again do this by forward reasoning (beginning at the branches), by backward reasoning (beginning at the common conclusion) or a mixture of both. It may use the proof context to store facts or it may use rules in the goal state, as described in the previous sections.

A forward reasoning step at a position where several branches join uses several current facts to prove a new current fact. Every forward reasoning step selects a subset of the stored local facts as the current facts and uses them to prove a new local fact from them.

A backward reasoning step may now produce several new current goals, which belong to different branches in the tree. A step always produces the goals for all branches which join at the current position in the tree. In this situation a single goal in the goal state is replaced by several goals. If rules are used as goals the assumptions from the old goal must be copied to all new goals. Facts stored in the proof context need not be copied since they are available for all goals. A proof is complete when it is complete for all goals in the goal state.

Proving from External Facts

The branches in the fact tree need not always start at an assumption A_i, they may also start at an "external" fact which is not part of the local proof context. In such cases the used external facts are referenced by their names. In that way a proof can use facts from the enclosing theory and a subproof can use facts from the enclosing proof(s) and the enclosing toplevel theory.

In particular, if the proposition of a theorem has no assumptions, i.e., the proposition is a formula and consists only of the conclusion `C`, every proof must start at one or more external facts (if `C` is no tautology which is valid by itself).

2.2.3 Basic Proof Structure

A proof is written in outer syntax and mainly describes how the fact tree is constructed which leads from the assumptions or external facts to the conclusion.

Statements and Methods

There are two possible operations for modifying the proof state: statements and method applications.

A statement adds one or more elements to the proof context. In particular, a statement may "state a fact", i.e., add a fact to the proof context, this is the reason for its name. A statement normally does not modify the goal state, there is one specific statement which may remove a goal from the goal state.

A method application modifies the goal state but normally leaves the proof context unchanged. The goal state is always modified so that, if all goals in the new state can be proved, then also all goals in the old state can be proved. This kind of goal state modification is also called a "refinement step".

When writing a proof the "proof mode" determines the kind of operation which may be written next: whether a statement (mode: `proof(state)`) or a method application (mode: `proof(prove)`) is admissible.

At the beginning of a proof the mode is always `proof(prove)`, i.e., a method application is expected. In the course of the proof it is possible to switch to mode `proof(state)` for entering statements, but not back again. After switching to statement mode the proof must be completed without further modifications to the goal state other than removing goals, only at the end a last method may be applied.

However, for every statement that states a fact a (sub-)proof must be specified, which again starts in mode `proof(prove)`. This way it is possible to freely switch between both modes in the course of a proof with nested subproofs.

A backward reasoning step always modifies the goal state, therefore it must be expressed by a method application. A forward reasoning step may be expressed by a statement, if intermediate facts are stored in the proof context. If intermediate facts are stored as assumptions in rules in the goal state, forward reasoning steps must also be expressed by method applications.

This implies that a sequence of statements can only represent a proof by forward reasoning where intermediate facts are stored in the proof context, whereas a sequence of method applications can represent an arbitrary proof where all facts are stored using rules in the goal state.

As described in Section 2.2.1 statements have the advantage that the facts added to the proof context are explicitly specified by the proof writer and are visible in the session document. That makes it easier to write and read a proof which consists only of statements. Method applications specify an operation on the goal state by name, the resulting new goal state is determined by Isabelle. It is visible for the proof writer in the interactive editor, but it is not visible in the session document for a reader of the proof. Therefore proofs consisting of method applications are difficult to understand and the proof writer must anticipate the effect of a method on the goal state when writing a proof step.

Proof Syntax

If $\mathtt{MA}_i$ denote method applications and $\mathtt{ST}_i$ denote statements, the general form of a proof is

$\mathtt{MA}_1 \ldots \mathtt{MA}_n$
proof $\mathtt{MA}_{n+1}$
  $\mathtt{ST}_1 \ldots \mathtt{ST}_m$
qed $\mathtt{MA}_{n+2}$

The last step $\mathtt{MA}_{n+2}$ may be omitted if it is not needed.

The part **proof** $\mathtt{MA}_{n+1}$ switches from method application mode `proof(prove)` to statement mode `proof(state)`.

The part **proof** … **qed** can be omitted and replaced by **done**, then the proof only consists of method applications and has the form $\mathtt{MA}_1 \ldots \mathtt{MA}_n$ **done**. Such proofs are called "proof scripts".

Since a proof script does not contain statements it cannot use the proof context to store facts. Proof scripts are intended to store facts as assumptions in the goal state or to apply only backward reasoning, where no intermediate facts need to be stored in addition to the goals (see Section 2.2.2).

If the method applications $\mathtt{MA}_1 \ldots \mathtt{MA}_n$ are omitted the proof only consists of the statements part and has the form

proof $\mathtt{MA}_1$
  $\mathtt{ST}_1 \ldots \mathtt{ST}_m$
qed $\mathtt{MA}_2$

where $\mathtt{MA}_2$ can also be omitted. Such proofs are called "structured proofs".

Since structured proofs consist nearly completely of statements, they are intended to use forward reasoning and store all assumptions and intermediate facts in the proof context.

A structured proof can be so simple, that it has no statements. For this case the syntax

by $\mathtt{MA}_1$ $\mathtt{MA}_2$

abbreviates the form **proof** $\mathtt{MA}_1$ **qed** $\mathtt{MA}_2$. Again, $\mathtt{MA}_2$ can be omitted which leads to the form

by $\mathtt{MA}_1$

In this form the proof consists of a single method application which directly leads from the conclusion `C` to the assumptions and used external facts.

As described in the previous section, a structured proof is usually easier to read and write than a proof script, since in the former case the sequence of the facts F_i is explicitly specified in the proof text, whereas in the latter case the sequence of the facts F_i is implicitly constructed and the proof text specifies only the methods.

However, since every statement for a forward reasoning step again requires a proof as its part (a "subproof" for the stated fact), no proof can be written using statements alone. The main idea of writing "good" proofs is to use nested structured proofs until every subproof is simple enough to be done in a single method application, i.e., the applied method directly goes from the conclusion to the assumption of the subproof. Such a simple proof can always be written in the form **by** `MA`.

Fake Proofs

A proof can also be specified as

sorry

This is a "fake proof" which turns the proposition to a fact without actually proving it.

A fake proof can be specified at any point in method application mode, so it can be used to abort a proof script in the form MA_1 ... MA_n **sorry**.

A structured proof in statement mode cannot be aborted in this way, however, subproofs can be specified as fake proofs. This makes it possible to interactively develop a structured proof in a top-down way, by first stating all required facts for the sequence from the assumptions to the goal with fake subproofs and then replacing the fake proofs by actual subproofs.

Nested Proof Contexts

Instead of a single proof context a proof may use a set of nested proof contexts, starting with an outermost proof context. In a nested context the content of the enclosing contexts is available together with the local content. When a nested context is ended, it is removed together with all its local content.

A nested proof context is created syntactically by enclosing statements in braces:

$ST_1 \ldots ST_m$ **{** $ST_{m+1} \ldots ST_n$ **}** $ST_{n+1} \ldots$

Note that according to the description until now the nested context is useless, because the facts introduced by its statements are removed at its end and cannot contribute to the proof. How the content of a nested context can be "exported" and preserved for later use will be explained further below.

For names (of fixed variables, facts and term abbreviations), nested contexts behave like a usual block structure: A name can be redefined in a nested context, then the named object in the outer context becomes inaccessible ("shadowed") in the inner context, but becomes accessible again when the inner context ends.

When two nested contexts follow each other immediately, this has the effect of "clearing" the content of the inner contexts: the content of the first context is removed and the second context starts being empty. This can be denoted by the keyword

next

which can be thought of being equivalent to a pair *}* *{* of adjacent braces.

Moreover the syntax **proof** $\mathtt{method}_1$ $\mathtt{ST}_1$ ... $\mathtt{ST}_n$ **qed** $\mathtt{method}_2$ automatically wraps the statements $\mathtt{ST}_1$ ... $\mathtt{ST}_n$ in a nested context. Therefore it is possible to denote a structured proof which only consists of a sequence of nested contexts without braces as

proof $\mathtt{method}_1$
  $\mathtt{ST}_{11}$ ... $\mathtt{ST}_{1m1}$ **next** $\mathtt{ST}_{21}$ ... $\mathtt{ST}_{2m2}$ **next** ... **next** $\mathtt{ST}_{n1}$ ... $\mathtt{ST}_{nmn}$
qed $\mathtt{method}_{n+2}$

where each occurrence of **next** clears the content of the inner context.

Another consequence of this wrapping is that no statement can add elements directly to the outermost proof context. The outermost proof context can only be filled by the initializations done by the structured theorem forms as described in Section 2.2.1. The resulting content of the context is not affected by clearing nested contexts and remains present until the end of the proof.

Also the goal state of a proof is not affected by the begin or end of a nested context. The goal state can be considered to be in the scope of the outermost context, it may use fixed variables from it. However, it is outside of all nested contexts and cannot contain elements from them.

Subproofs

A subproof takes a single goal and solves it as part of another proof. It has its own goal state which hides the goal state of the enclosing proof until the subproof is complete.

The outermost proof context used by the subproof is nested in the context of the enclosing proof, therefore all content of the enclosing proof context is available there and can be referenced by name, as long as the name is not shadowed by a redefinition in the subproof.

There are mainly two kinds of subproofs: proving a goal which is already in the goal state of the enclosing proof or specifying a new goal which becomes a fact in the proof context after proving it.

The first kind of subproof has the form

subgoal ⟨*proof*⟩

and may be specified in method application mode `proof(prove)` in place of a single method application. Its initial goal state contains a copy of the first goal from the goal state of the enclosing proof. If the subproof is successfully terminated it removes that goal from the goal state of the enclosing proof.

There is also the "structured form"

subgoal premises `name` ⟨*proof*⟩

If the goal processed by the subproof is a derivation rule $[\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow C$ it takes the assumptions $A_1,\ \ldots,\ A_n$, and adds them as assumed facts into the outermost proof context of the subproof, like the structured forms of theorems do for their assumptions (see Section 2.2.1). If `name` is specified it is used for naming the set of assumptions, if it is omitted the default name `prems` is used.

Using this form a subproof can be written as a structured proof which stores its facts in its proof context, although the enclosing proof is a proof script and stores its facts as rule assumptions in the goal state.

The second kind of subproof occurs as part of some statements, as described in Section 2.2.5.

2.2.4 Method Application

A method application mainly specifies a proof method to be applied.

Proof Methods

Proof methods are basically denoted by method names, such as `standard`, `simp`, or `rule`. A proof method name can also be a symbol, such as `-`.

A method may have arguments, then it usually must be delimited by parentheses such as in `(rule example1)` or `(simp add: example2)`, where `example1` and `example2` are fact names.

Isabelle supports a large number of proof methods. A selection of proof methods used in this manual is described in Section 2.3.

Method Application

A standalone method application step is denoted as

apply `method`

where `method` denotes the proof method to be applied, together with its arguments.

The method applications which follow **proof** and **qed** in a structured proof are simply denoted by the applied method. Hence the general form of a proof is

apply $\mathtt{method}_1$ ... **apply** $\mathtt{method}_n$
proof $\mathtt{method}_{n+1}$
  $\mathtt{ST}_1$ ... $\mathtt{ST}_m$
qed $\mathtt{method}_{n+2}$

where $\mathtt{ST}_1$ ... $\mathtt{ST}_m$ are statements. The method $\mathtt{method}_{n+1}$ is called the “initial method” of the structured proof part.

2.2.5 Stating Facts

The most basic kind of statements is used to add a fact to the proof context.

Adding a Fact to the Proof Context

A fact is added to the (innermost enclosing) proof context by a statement of the form

have `"prop"` ⟨*proof*⟩

where `prop` is a proposition in inner syntax and ⟨*proof*⟩ is a (sub-) proof for it. This form is similar to the specification of a theorem in a theory and has a similar effect in the local proof context.

As for a theorem the fact can be named:

have `name: "prop"` ⟨*proof*⟩

The scope of the name is the innermost proof context enclosing the statement. In their scope named facts can be displayed and searched as described for theorems in Section 2.1.7.

As for a theorem, if the fact is a derivation rule it may also be specified in structured form:

have `"C"` **if** $\mathtt{"A}_1\mathtt{"}$ ... $\mathtt{"A}_n\mathtt{"}$ **for** x_1 ... x_m ⟨*proof*⟩

where the assumptions and variables may be grouped by **and**, assumption groups may be named, and a type may be specified for each variable group.

Note, however, that the structured form using **fixes**, **assumes**, and **shows** (see Section 2.1.7) is not available for stating facts in a proof.

The **have** statement is called a “goal statement”, because it states the proposition *prop* as a (local) goal which is then proved by the subproof ⟨*proof*⟩.

Note that the name given to the fact to be proved cannot be used to access it in the subproof, because it is only assigned after the proof has been finished, whereas names given to assumption groups can only be used in the subproof because their scope is the proof context of the subproof.

Proving a Goal

A proof using only forward reasoning ends, if the last stated fact F_n unifies with the conclusion C. If the facts are stored in the proof context, F_n will be added by a statement. Therefore a special form of stating a fact exists, which, after proving the fact, tries to unify it with a goal in the goal state of the enclosing proof, and, if successful, removes the goal from that goal state. This is done by the statement

show *"prop"* ⟨*proof*⟩

which is the only statement which may affect the goal state of the enclosing proof. Its syntax is the same as for **have**, including naming and structured form. Like **have** it is also called a “goal statement”.

As described in Section 2.2.3, statements are always wrapped by a nested proof context. When the **show** statement tries to unify its fact with a goal from the goal state it replaces all variables fixed in an enclosing nested proof context by unknowns (which is called “exporting the fact” from the proof context) so that they can match arbitrary subterms (of the correct type) in the goal.

If the unification of the exported fact with some goal is not successful the step is erroneous and the proof cannot be continued, in the interactive editor an error message is displayed.

If a goal has the form of a derivation rule, the exported fact is only unified with the conclusion of the goal. If also the exported fact is a derivation rule, additionally each of its assumptions must unify with an assumption of the goal.

This is a special case of a refinement step in the sense of Section 2.2.3. Whenever the exported fact can be proved, also the matching goal can be proved. Since the exported fact has just been proved by the subproof, the matching goal has been proved as well and may be removed from the enclosing goal state. So the condition for a successful **show** statement can be stated as “the exported fact must refine a goal” (this term is used in error messages).

Note that in a proof using only forward reasoning the proposition *prop* in a **show** statement is the same proposition which has been specified as conclusion C in the proposition $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ which shall be proved by the proof. To avoid repeating it, Isabelle automatically provides the term abbreviation `?thesis` for it in the outermost proof context. So in the simplest case the last step of a structured proof by forward reasoning can be written as

show `?thesis` ⟨*proof*⟩

The abbreviation `?thesis` is a single identifier, therefore it needs not be quoted.

If, however, the application of the initial method `method` in a structured proof **proof** `method` ... modifies the original goal, this modification is not reflected in `?thesis`. So a statement **show** `?thesis` ⟨*proof*⟩ will usually not work, because `?thesis` no more refines the modified goal. Instead, the proof writer must know the modified goal and specify it explicitly as proposition in the **show** statement. If the `method` splits the goal into several new goals, several **show** statements may be needed to remove them.

To test whether a proposition refines a goal in the enclosing goal state, a **show** statement can be specified with a fake proof:

show `"prop"` **sorry**

If that statement is accepted, the proposition refines a goal and removes it.

2.2.6 Facts as Proof Input

If a linear fact sequence F_1, ..., F_n where every fact implies the next one is constructed in statement mode in the form

have "F_1" ⟨*proof*⟩$_1$
...
have "F_n" ⟨*proof*⟩$_n$

every fact F_i needs to refer to the previous fact F_{i-1} in its proof ⟨*proof*⟩$_i$. This can be done by naming all facts

have $name_1$: "F_1" ⟨*proof*⟩$_1$
...
have $name_n$: "F_n" ⟨*proof*⟩$_n$

and refer to F_{i-1} in $proof_i$ by $name_{i-1}$.

Isabelle supports an alternative way by passing facts as input to a proof.

Using Input Facts in a Proof

The input facts are passed as input to the first method applied in the proof. In a proof script it is the method applied in the first **apply** step, in a structured proof **proof** `method` ... it is the initial method `method`.

Every proof method accepts a set of facts as input. Whether it processes them and how it uses them depends on the kind of method. Therefore input facts for a proof only work in the desired way, if a corresponding method is used at the beginning of the proof. See Section 2.3 for descriptions how methods process input facts.

Inputting Facts into a Proof

In method application mode *proof(prove)* facts can be input to the remaining proof ⟨*proof*⟩ by

using *name*$_1$ ... *name*$_n$ ⟨*proof*⟩

where the *name*$_i$ are names of facts or fact sets. The sequence of all referred facts is input to the proof following the **using** specification. In a proof script it is input to the next **apply** step. If a structured proof follows, it is input to its initial method. Since in method application mode no local facts are stated by previous steps, the facts can only be initial facts in the outermost proof context (see Section 2.2.1), or they may be external facts from the enclosing theory, or, if in a subproof, they may be facts from contexts of enclosing proofs.

In statement mode *proof(state)* fact input is supported with the help of the special fact set name *this*. The statement

then

inputs the facts named *this* to the proof of the following goal statement.

The statement **then** must be immediately followed by a goal statement (**have** or **show**). This is enforced by a special third proof mode *proof(chain)*. In it only a goal statement is allowed, **then** switches to this mode, the goal statement switches back to mode *proof(state)* after its proof.

Note that **then** is allowed in statement mode, although it does not state a fact. There are several other such auxiliary statements allowed in mode *proof(state)* in addition to the goal statements **have** and **show**.

The fact set *this* can be set by the statement

note *name*$_1$ ... *name*$_n$

Therefore the statement sequence

note *name*$_1$ ... *name*$_n$
then **have** *"prop"* ⟨*proof*⟩

inputs the sequence of all facts referred by *name*$_1$ ... *name*$_n$ to the ⟨*proof*⟩, in the same way as **using** inputs them to the remaining proof following it.

The statement sequence

note *name*$_1$ ... *name*$_n$ **then**

can be abbreviated by the statement

from *name*$_1$ ... *name*$_n$

Like **then** it switches to mode `proof(chain)` and it inputs the sequence of the facts referred by *name*$_1$ ... *name*$_n$ to the proof of the following goal statement.

2.2.7 Fact Chaining

In both cases described for fact input until now, the facts still have been referred by names. This can be avoided by automatically using a stated fact as input to the proof of the next stated fact. That is called "fact chaining".

Automatic Update of the Current Facts

Fact chaining is achieved, because Isabelle automatically updates the fact set `this`. Whenever a new fact is added to the proof context, the set `this` is redefined to contain (only) this fact. In particular, after every goal statement `this` names the new proved fact. Therefore the fact set `this` is also called the "current facts".

Thus a linear sequence of facts where each fact implies the next one can be constructed by

have "F_1" $\langle proof\rangle_1$
then have "F_2" $\langle proof\rangle_2$
...
then have "F_n" $\langle proof\rangle_n$

Now in every *proof*$_i$ the fact F_{i-1} is available as input and can be used to prove F_i.

In this way a structured proof for a rule $A \Longrightarrow C$ can be written which uses a fact sequence $A,\ F_2,\ \ldots\ F_{n-1},\ C$. If the theorem is specified in the structured form **theorem** "C" **if** "A" which adds the assumption to the proof context and names it `that` (see Section 2.2.1) the proof consists of the statement sequence

from *that* **have** "F_2" $\langle proof\rangle_1$
then have "F_3" $\langle proof\rangle_2$
...
then have "F_{n-1}" $\langle proof\rangle_2$
then show *?thesis* $\langle proof\rangle_n$

For the structured form **theorem assumes** "A" **shows** "C" the assumption name `assms` must be used instead of `that`.

Chaining can be combined with explicit fact referral by a statement of the form

note *this* *name*$_1$ ... *name*$_n$

It sets *this* to the sequence of *this* and the *name*$_1$ ... *name*$_n$, i.e., it adds the *name*$_1$ ... *name*$_n$ to *this*. In this way the current facts can be extended with other facts and then chained to the proof of the next stated fact.
The statement sequence

note *this* *name*$_1$ ... *name*$_n$ **then**

can be abbreviated by the statement

with *name*$_1$ ... *name*$_n$

Like **then** it switches to mode *proof(chain)* and it inputs the sequence of the facts referred by *name*$_1$ ... *name*$_n$ together with the current facts to the proof of the following goal statement.
If a proof consists of a fact tree with several branches, every branch can be constructed this way. Before switching to the next branch the last fact must be named, so that it can later be used to prove the fact where the branches join. A corresponding proof pattern for two branches which join at fact *F* is

have "F_{11}" $\langle proof\rangle_{11}$
then have "F_{12}" $\langle proof\rangle_{12}$
...
then have *name*$_1$*:* "F_{1m}" $\langle proof\rangle_{1m}$
have "F_{21}" $\langle proof\rangle_{21}$
then have "F_{22}" $\langle proof\rangle_{22}$
...
then have "F_{2n}" $\langle proof\rangle_{2n}$
with *name*$_1$ **have** "F" $\langle proof\rangle$

If the theorem has been specified in structured form **theorem** "*C*" **if** "A_1" ... "A_n" every branch can be started in the form

from *that* **have** "F_{11}"
...

which will input all assumptions to every branch. This works since unneeded assumptions usually do not harm in a proof, but it is often clearer for the reader to explicitly name the assumptions

theorem "*C*" **if** a_1*:* "A_1" **and** ... **and** a_n*:* "A_n"

and specify only the relevant assumption by name in the proof:

from a_1 **have** "F_{11}"
...

Naming and Grouping Current Facts

Since the fact set built by a **note** statement is overwritten by the next stated fact, it is possible to give it an explicit name in addition to the name *this* in the form

note *name* = $\textit{name}_1$ ... $\textit{name}_n$

The *name* can be used later to refer to the same fact set again, when *this* has already been updated. Defining such names is only possible in the **note** statement, not in the abbreviated forms **from** and **with**.

The facts specified in **note**, **from**, **with**, and **using** can be grouped by separating them by **and**. Thus it is possible to write

from $\textit{name}_1$ **and** ... **and** $\textit{name}_n$ **have** *"prop"* ⟨*proof*⟩

In the case of a **note** statement every group can be given an additional explicit name as in

note $\textit{name}_1$ = $\textit{name}_{11}$ ... $\textit{name}_{1m1}$ **and** ... **and** $\textit{name}_n$ = $\textit{name}_{n1}$ ... $\textit{name}_{nmn}$

Accessing Input Facts in a Proof

At the beginning of a proof the set name *this* is undefined, the name cannot be used to refer to the input facts (which are the current facts in the enclosing proof). To access the input facts other than by the first proof method they must be named before they are chained to the goal statement, then they can be referenced in the subproof by that name. For example in

note *input* = *this*
then **have** *"prop"* ⟨*proof*⟩

the input facts can be referenced by the name *input* in ⟨*proof*⟩.

Exporting the Current Facts of a Nested Context

At the end of a nested context (see Section 2.2.3) the current facts are automatically exported to the enclosing context, i.e. they become available there as the fact set named *this*, replacing the current facts existing before the nested context. This is another way how facts from a nested context can contribute to the overall proof.

Basically, only the last fact is current at the end of a context. Arbitrary facts can be exported from the nested context by explicitly making them current at its end, typically using a **note** statement:

```
... {
  have f₁: "prop₁" ⟨proof⟩₁
  ...
  have fₙ: "propₙ" ⟨proof⟩ₙ
  note f₁ ... fₙ
  } ...
```

Here all facts are named and the **note** statement makes them current by referring them by their names. Note, that the names are only valid in the nested context and cannot be used to refer to the exported facts in the outer context.

The exported facts can be used in the outer context like all other current facts by directly chaining them to the next stated fact:

```
... { ... } then have "prop" ⟨proof⟩  ...
```

or by naming them for later use, with the help of a **note** statement:

```
... { ... } note name = this ...
```

2.2.8 Assuming Facts

Facts can be added to the proof context without proving them, then they are only assumed.

Introducing Assumed Facts

A proposition is inserted as assumption in the proof context by a statement of the form

```
assume "prop"
```

Several assumptions can be inserted in a single **assume** statement of the form

```
assume "prop₁" ... "propₙ"
```

As usual, the assumptions can be grouped by **and** and the groups can be named. In their scope named assumptions can be displayed and searched like other named facts (see Section 2.1.7).

Assumed facts may be derivation rules, then they may be specified directly as proposition or in structured form

```
assume "C" if "A₁" ... "Aₙ" for x₁ ... xₘ
```

The rule assumptions A_1, ..., A_n may be grouped by **and**, however, the groups cannot be named since there is no subproof where the names could be used. Note that variables occurring in the propositions C, A_1, ..., A_n are only turned to unknowns if the are explicitly bound in the **for** part, otherwise they refer to variables bound in an enclosing proof context or remain free in the assumed rule (which is usually an error).

If several rules are specified in the same statement there is only one **if** and **for** part common for all rules. Every rule gets a copy of the assumptions from the **if** part.

Like goal statements an **assume** statement makes the assumed facts current, i.e. it updates the set *this* to contain the specified propositions as facts, so that they can be chained to a following goal statement:

assume "A"
then have "*prop*" ⟨*proof*⟩
...

Exporting Facts with Assumptions

Assumed facts may be used to prove other local facts, so that arbitrary local facts may depend on the validness of the assumed facts. This is taken into account when local facts are exported from a proof context (see Section 2.2.5). Whenever a local fact F is exported it is combined with copies of all locally assumed facts $AF_1,\ldots,AF_n$ to the derivation rule $[\![AF_1;\ldots;AF_n]\!] \Longrightarrow F$, so that F still depends on the assumptions after leaving the context.

If the fact F is itself a derivation rule $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ then the locally assumed facts are prepended, resulting in the exported rule $[\![AF_1;\ldots;AF_n;A_1;\ldots;A_n]\!] \Longrightarrow C$.

If the fact F has been proved in a **show** statement it is also exported in this way, resulting in a derivation rule with all local assumptions added. Therefore it will only refine a goal if every local assumption unifies with an assumption present in the goal, (see Section 2.2.5).

If in a previous part of a proof facts have been stored as rule assumptions in the goal state (see Section 2.2.2), they can be "copied" to the proof context with the help of **assume** statements and will be "matched back" by the **show** statements used to remove the goals.

In particular, if a theorem specifies a rule $A \Longrightarrow C$ directly as proposition it will become the initial goal, as described in Section 2.2.1. Then a structured proof using the fact sequence A, F_2, ..., F_{n-1}, C can be written

assume "A"
then have "F_2" ⟨*proof*⟩
...

then have *"F_{n-1}"* ⟨*proof*⟩
then show *?thesis* ⟨*proof*⟩

The **show** statement will export the rule *A* $\Longrightarrow$ *C* which matches and removes the goal.

Presuming Facts

It is also possible to use a proposition as assumed fact which does not unify with an assumption in a goal, but can be proved from them. In other words, the proof is started somewhere in the middle of the fact tree, works by forward reasoning, and when it reaches the conclusion the assumed fact remains to be proved. The statement

presume *"prop"*

inserts such a presumed fact into the proof context. Like for **assume** several assumptions may be specified and the structured form with **if** and **for** is supported.

When a fact is exported from a context with presumed facts, they do not become a part of the exported rule. Instead, at the end of the context for each presumed fact F_p a new goal $[\![A_1;\ldots;A_n]\!] \Longrightarrow F_p$ is added to the goal state of the enclosing proof where $A_1,\ldots,A_n$ are the facts assumed in the ended context. So the proof has to continue after proving all original goals and is only finished when all such goals for presumed facts have been proved as well.

2.2.9 Fixing Variables

Variables which have not been declared or defined as a constant in the enclosing theory are called "free" if they occur in the proposition of a theorem. Such variables are automatically added as fixed variables to the outermost proof context, thus they can be used everywhere in the proof where they are not shadowed. If, instead, they are explicitly bound in the proposition (see Section 2.1.6), their use is restricted to the proposition itself. Thus in

theorem *"⋀x::nat. x < 3 $\Longrightarrow$ x < 5"* ⟨*proof*⟩

the variable *x* is restricted to the proposition and is not accessible in ⟨*proof*⟩, whereas in

theorem *"(x::nat) < 3 $\Longrightarrow$ x < 5"* ⟨*proof*⟩

the variable *x* is accessible in ⟨*proof*⟩. If the theorem is specified in structured form, variables may be explicitly specified to be fixed in the outermost proof context using **for** or **fixes**, respectively (see Section 2.2.1). Therefore the forms

theorem "$x < 3 \Longrightarrow x < 5$" **for** x::nat ⟨*proof*⟩

and

theorem fixes x::nat **shows** "$x < 3 \Longrightarrow x < 5$" ⟨*proof*⟩

are completely equivalent to the previous form. Thus explicitly fixing variables is optional for theorems, however it usually improves readability and provides a place where types can be specified for them in a systematic way.

Local Variables

Additional variables can be added as fixed to the innermost enclosing proof context by the statement

fix $x_1 \ldots x_n$

As usual the variables can be grouped by **and** and types can be specified for (some of) the groups.

A fixed variable in a proof context denotes a specific value of its type. However, if the variable has been introduced by **fix**, **for** or **fixes** it is underspecified in the same way as a constant introduced by **consts** in a theory (see Section 2.1.3): no information is given about its value. In this sense it denotes an "arbitrary but fixed value".

If a variable name is used in a proof context without explicitly fixing it, it either refers to a variable in an enclosing context or it is free. If it is explicitly fixed it names a variable which is different from all variables with the same name in enclosing contexts.

If a variable is free in the proof for a theorem its value cannot be proved to be related in any way to values used in the theorem. Therefore it is useless for the proof and its occurrence should be considered to be an error. In the interactive editor Isabelle marks free variables by a light red background as an alert for the proof writer.

A fixed local variable is common to the whole local context. If it occurs in several local facts it always is the same variable and denotes the same value, it is not automatically restricted to every fact, as for toplevel theorems. Hence in

fix x::nat
assume a: "$x < 3$"
have "$x < 5$" ⟨*proof*⟩

the ⟨*proof*⟩ may refer to fact a because the x is the same variable in both facts.

Exporting Facts with Local Variables

Explicitly fixing variables in a proof context is not only important for avoiding name clashes. If a fact is exported from a proof context, all fixed local variables are replaced by unknowns, other variables remain unchanged. Since unification only works for unknowns, it makes a difference whether a fact uses a local variable or a variable which origins from an enclosing context or is free.

The proposition `x < 3` $\Longrightarrow$ `x < 5` can be proved by the statements

fix *y::nat*
assume *"y < 3"*
then show *"y < 5"* ⟨*proof*⟩

because when the fact `y < 5` is exported, the assumption is added (as described in Section 2.2.8) and then variable `y` is replaced by the unknown `?y` because `y` has been locally fixed. The result is the rule `?y<3` $\Longrightarrow$ `?y<5` which unifies with the proposition.

If, instead, `y` is not fixed, the sequence

assume *"(y::nat) < 3"*
then have *"y < 5"* ⟨*proof*⟩

still works and the local fact `y < 5` can be proved, but it cannot be used with the **show** statement to prove the proposition `x < 3` $\Longrightarrow$ `x < 5`, because the exported rule is now `y<3` $\Longrightarrow$ `y<5` which does not unify with the proposition, it contains a different variable instead of an unknown.

If variables are explicitly bound in the proposition of a theorem they are not accessible in the proof. Instead, corresponding new local variables (which may have the same names) must be fixed in the proof context and used for the proof. Upon export by a **show** statement these variables will be replaced by unknowns which then are unified with the variables in the goal. A corresponding proof for the proposition $\bigwedge$`x::nat. x < 3` $\Longrightarrow$ `x < 5` is

fix *y::nat*
assume *"y < 3"*
then show *"y < 5"* ⟨*proof*⟩

2.2.10 Obtaining Variables

Local variables may also be introduced together with a fact which allows to specify additional information about their value. This is done using a statement of the form

obtain x_1 ... x_m **where** *"prop"* ⟨*proof*⟩

where *prop* is a proposition in inner syntax which contains the variables x_1 ... x_m. Like for variables introduced by **fix** the variables can be grouped by **and** and types can be specified for (some of) the groups.

The proposition usually relates the values of the new variables to values of existing variables (which may be local or come from enclosing contexts). In the simplest case the proposition directly specifies terms for the new variables, such as in

fix *x::nat*
obtain *y z* **where** *"y = x + 3 ∧ z = x + 5"* ⟨*proof*⟩

But it is also possible to specify the values indirectly:

fix *x::nat*
obtain *y z* **where** *"x = y - 3 ∧ y + z = 2*x +8"* ⟨*proof*⟩

Here the proposition may be considered to be an additional assumption which is added to the proof context.

Proving *obtain* Statements

Actually, several propositions may be specified in an **obtain** statement:

obtain x_1 ... x_m **where** *"*$prop_1$*"* ... *"*$prop_n$*"* ⟨*proof*⟩

The propositions may be grouped by **and** and the groups can be named as usual. This **obtain** statement has a similar meaning as the statements

fix x_1 ... x_m
assume *"*$prop_1$*"* ... *"*$prop_n$*"*

but there is one important difference: the propositions in an **obtain** statement must be redundant in the local proof context.

That is the reason why an **obtain** statement is a goal statement and includes a proof. The proof must prove the redundancy of the propositions, which may be stated in the following way: if any other proposition can be derived from them in the local proof context it must be possible to also derive it without the propositions. This can be stated formally as

$(\bigwedge x_1 \ldots x_m.\ [\![prop_1; \ldots; prop_n]\!] \Longrightarrow P) \Longrightarrow P$

which is exactly the goal to be proved for the **obtain** statement.

Consider the statements

fix *x::nat*
obtain *y* **where** *"x = 2*y"* ⟨*proof*⟩

This proposition is not redundant, because it implies that *x* must be even. Therefore no proof exists.

Note that after a successful proof of an **obtain** statement the current facts are the propositions specified in the statement, not the proved redundancy statement. Input facts may be passed to **obtain** statements. Like for the other goal statements, they are input to the ⟨*proof*⟩.

Exporting Facts after Obtaining Variables

Unlike facts assumed by an **assume** statement (see Section 2.2.8) the propositions in an **obtain** statement are *not* added as assumptions when a fact *F* is exported from the local context. This is correct, since they have been proved to be redundant, therefore they can be omitted.

However, that implies that an exported fact *F* may not refer to variables introduced by an **obtain** statement, because the information provided by the propositions about them gets lost during the export.

2.2.11 Term Abbreviations

A term abbreviation is a name for a proposition or a term in it.

Defining Term Abbreviations

A term abbreviation can be defined by a statement of the form

let *?name = "term"*

Afterwards the name is "bound" to the term and can be used in place of the term in propositions and other terms, as in:

let *?lhs = "2*x+3"*
let *?rhs = "2*5+3"*
assume *"x < 5"*
have *"?lhs ≤ ?rhs"* ⟨*proof*⟩

The name *?thesis* (see Section 2.2.5) is a term abbreviation of this kind. It is defined automatically for every proof in the outermost proof context.

A **let** statement can define several term abbreviations in the form

let *?name*$_1$ = *"term*$_1$*"* **and** ... **and** *?name*$_n$ = *"term*$_n$*"*

A **let** statement can occur everywhere in mode *proof(state)*. However, it does not preserve the current facts, the fact set *this* becomes undefined by it.

Pattern Matching

Note that term abbreviations have the form of "unknowns" (see Section 2.1.7), although they are defined ("bound"). The reason is that they are actually defined by unification.

The more general form of a **let** statement is

```
let "pattern" = "term"
```

where *pattern* is a term which may contain unbound unknowns. As usual, if the pattern consists of a single unknown, the quotes may be omitted. The **let** statement unifies *pattern* and *term*, i.e., it determines terms to substitute for the unknowns, so that the pattern becomes syntactically equal to *term*. If that is not possible, an error is signaled, otherwise the unknowns are bound to the substituting terms. Note that the equals sign belongs to the outer syntax, therefore both the pattern and the term must be quoted separately.

The **let** statement

```
let "?lhs ≤ ?rhs" = "2*x+3 ≤ 2*5+3"
```

binds the unknowns to the same terms as the two **let** statements above.

Pattern matching can be used to define parameterized abbreviations. If the pattern has the form of a function application where the unknown is the function, it will be bound to a function which, after substituting the arguments, will be equal to the *term*. So it can be applied to other arguments to yield terms where the corresponding parts have been replaced. This kind of pattern matching is called "higher order unification" and only succeeds if the pattern and *term* are not too complex.

The **let** statement

```
let "?lhs x ≤ ?rhs 5" = "2*x+3 ≤ 2*5+3"
```

binds both unknowns to the lambda term λa. 2 * a + 3. Thus afterwards the use ?lhs 7 results in the term 2 * 7 + 3.

The *term* may contain unknowns which are already bound. They are substituted by their bound terms before the pattern matching is performed. Thus the term can be constructed with the help of abbreviation which have been defined previously. A useful example is matching a pattern with ?thesis:

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5+3"
proof method
  let "?lhs ≤ ?rhs" = ?thesis
  ...
```

to reuse parts of the conclusion in the proof without specifying them explicitly.

Note that the unknowns are only bound at the end of the whole **let** statement. In the form

```
let "pattern₁" = "term₁" and ... and "patternₙ" = "termₙ"
```

the unknowns in $pattern_i$ cannot be used to build $term_{i+1}$ because they are not yet bound. In contrast, in the sequence of **let** statements

```
let "pattern₁" = "term₁"
 ...
let "patternₙ" = "termₙ"
```

the unknowns in $pattern_i$ can be used to build $term_{i+1}$ because they are already bound.

If a bound unknown occurs in the pattern its bound term is ignored and the unknown is rebound according to the pattern matching. In particular, it does not imply that the old and new bound terms must be equal, they are completely independent.

If a part of the term is irrelevant and need not be bound the dummy unknown "_" (underscore) can be used to match it in the pattern without binding an unknown to it:

```
let "_ ≤ ?rhs" = "2*x+3 ≤ 2*5+3"
```

will only bind *?rhs* to the term on the righthand side.

If the term internally binds variables which are used in a subterm, the subterm cannot be matched separately by an unknown because then the variable bindings would be lost. Thus the statement

```
let "λx. ?t" = "λx. x+1"
```

will fail to bind *?t* to *x+1* whereas

```
let "λx. x+?t" = "λx. x+1"
```

will successfully bind *?t* to *1* since the bound variable *x* does not occur in it.

Casual Term Abbreviations

Term abbreviations may also be defined in a "casual way" by appending a pattern to a proposition which is used for some other purpose in the form

```
"prop" (is "pattern")
```

The pattern is matched with the proposition *prop* and if successful the unknowns in the pattern are bound as term abbreviations.

This is possible for all propositions used in a theorem (see Section 2.1.7), such as in

```
theorem "prop" (is ?p) ⟨proof⟩
theorem fixes x::nat and c and n
  assumes asm1: "x < c" (is ?a1) and "n > 0" (is ?a2)
  shows "n*x < n*c" (is "?lhs < ?rhs")
  ⟨proof⟩
```

and also in the structured form using **if** (see Section 2.1.6). The unknowns will be bound as term abbreviations in the outermost proof context in the proof of the theorem.

Note the difference between the fact name `asm1` and the term abbreviation name `?a1` in the example. The fact name refers to the proposition `x < c` as a valid fact, it can only be used to work with the proposition as a logical entity. The abbreviation name `?a1`, instead, refers to the proposition as a syntactic term of type `bool`, it can be used to construct larger terms such as in `?a1 ∧ ?a2` which is equivalent to the term `x < c ∧ n > 0`.

Casual term abbreviations may also be defined for propositions used in goal statements (see Sections 2.2.5 and 2.2.10) and in **assume** and **presume** statements (see Section 2.2.8). Then the unknowns will be bound as term abbreviations in the *enclosing* proof context, so that they are available after the statement (and also in the nested subproof of the goal statements).

2.2.12 Accumulating Facts

Instead of giving individual names to facts in the proof context, facts can be collected in named fact sets. Isabelle supports the specific fact set named `calculation` and provides statements for managing it.

The fact set `calculation` is intended to accumulate current facts for later use. Therefore it is typically initialized by the statement

```
note calculation = this
```

and afterwards it is extended by several statements

```
note calculation = calculation this
```

After the last extension the facts in the set are chained to the next proof:

```
note calculation = calculation this then have ...
```

Support for Fact Accumulation

Isabelle supports this management of `calculation` with two statements. The statement

```
moreover
```

is equivalent to **note** `calculation = this` when it occurs the first time in a context, afterwards it behaves like **note** `calculation = calculation this` but without making `calculation` current, instead, it leaves the current fact(s) unchanged. The statement

ultimately

is equivalent to **note** `calculation = calculation this` **then**, i.e., it adds the current facts to the set, makes the set current, and chains it to the next goal statement.

Due to the block structure of nested proof contexts, the `calculation` set can be reused in nested contexts without affecting the set in the enclosing context. The first occurrence of **moreover** in the nested context initializes a fresh local `calculation` set. Therefore fact accumulation is always local to the current proof context.

Accumulating Facts in a Nested Context

Fact accumulation can be used for collecting the facts in a nested context for export (see Section 2.2.7) without using explicit names for them:

. . . **{**
  have "$prop_1$" $\langle proof\rangle_1$
  moreover have "$prop_2$" $\langle proof\rangle_2$
  . . .
  moreover have "$prop_n$" $\langle proof\rangle_n$
  moreover note `calculation`
  } . . .

Accumulating Facts for Joining Branches

Fact accumulation can also be used for collecting the facts at the end of joined fact branches in a proof and inputting them to the joining step. A corresponding proof pattern for two branches which join at fact F is

have "F_{11}" $\langle proof\rangle_{11}$
then have "F_{12}" $\langle proof\rangle_{12}$
. . .
then have "F_{1m}" $\langle proof\rangle_{1m}$
moreover have "F_{21}" $\langle proof\rangle_{21}$
then have "F_{22}" $\langle proof\rangle_{22}$
. . .
then have "F_{2n}" $\langle proof\rangle_{2n}$
ultimately have "F" $\langle proof\rangle$

The **moreover** statement starts the second branch and saves the fact F_{1m} to *calculation*. The **ultimately** statement adds the fact F_{2n} to *calculation* and then inputs the set to the proof of F.

Note that **moreover** does not chain the current facts to the following goal statement.

Using nested contexts sub-branches can be constructed and joined in the same way.

2.2.13 Equational Reasoning

Often informal proofs on paper are written in the style

$2*(x+3) = 2*x+6 \leq 3*x+6 < 3*x+9 = 3*(x+3)$

to derive the fact $2*(x+3) < 3*(x+3)$. Note that the formula shown above is not a wellformed proposition because of several occurrences of the toplevel relation symbols $=$, $\leq$ and $<$. Instead, the formula is meant as an abbreviated notation of the fact sequence

$2*(x+3) = 2*x+6,\ 2*x+6 \leq 3*x+6,\ 3*x+6 < 3*x+9,\ 3*x+9 = 3*(x+3)$

which sketches a proof for $2*(x+3) < 3*(x+3)$. This way of constructing a proof is called “equational reasoning” which is a specific form of forward reasoning.

Transitivity Rules

The full example proof needs additional facts which must be inserted into the sequence. From the first two facts the fact $2*(x+3) \leq 3*x+6$ is derived, then with the third fact the fact $2*(x+3) < 3*x+9$ is derived, and finally with the fourth fact the conclusion $2*(x+3) < 3*(x+3)$ is reached. The general pattern of these additional derivations can be stated as the derivation rules $[\![a = b;\ b \leq c]\!] \Longrightarrow a \leq c$, $[\![a \leq b;\ b < c]\!] \Longrightarrow a < c$, and $[\![a < b;\ b = c]\!] \Longrightarrow a < c$.

Rules of this form are called “transitivity rules”. They are valid for relations such as equality, equivalence, orderings, and combinations thereof.

This leads to the general form of a proof constructed by equational reasoning: every forward reasoning step starts at a fact F_i of the form $a\ r\ b$ where r is a relation symbol. It proves an intermediate fact H_i of the form $b\ r\ c$ where r is the same or another relation symbol and uses a transitivity rule to prove the fact F_{i+1} which is $a\ r\ c$. In this way it constructs a linear sequence of facts which leads to the conclusion.

The intermediate facts H_i are usually proved from assumptions or external facts, or they may have a more complex proof which forms an own fact branch which ends at H_i and is joined with the main branch at F_{i+1} with the help of the transitivity rule.

Support for Equational Reasoning

Isabelle supports equational reasoning in the following form. It provides the statement

also

which expects that the set *calculation* contains the fact F_i and the current fact *this* is the fact H_i. It automatically selects an adequate transitivity rule, uses it to derive the fact F_{i+1} and replaces F_i in *calculation* by it. Upon its first use in a proof context **also** simply stores the current fact *this* in *calculation*. The statement

finally

behaves in the same way but additionally makes the resulting fact F_{i+1} current, i.e., puts it into the set *this*, and chains it into the next goal statement. In other words, **finally** is equivalent to **also from** *calculation*.

Note that **also** behaves like **moreover** and **finally** behaves like **ultimately**, both with additional application of the transitivity rule.

Additionally, Isabelle automatically maintains the term abbreviation ... (which is the three-dot-symbol available for input in the Symbols panel (see Section 1.2.5) of the interactive editor in tab "Punctuation") for the term on the right hand side of the current fact. Together, the example equational reasoning proof from above can be written

```
have "2*(x+3) = 2*x+6" ⟨proof⟩
also have "... ≤ 3*x+6" ⟨proof⟩
also have "... < 3*x+9" ⟨proof⟩
also have "... = 3*(x+3)" ⟨proof⟩
finally show ?thesis ⟨proof⟩
```

where *?thesis* abbreviates the conclusion *2*(x+3) < 3*(x+3)*. This form is quite close to the informal paper style of the proof.

Determining Transitivity Rules

To automatically determine the transitivity rule used by **also** or **finally**, Isabelle maintains the dynamic fact set (see Section 2.1.7) named *trans*. It selects a rule from that set according to the relation symbols used in the facts in *calculation* and *this*.

A transitivity rule which is not in *trans* can be explicitly specified by name in the form

also *(name)*
finally *(name)*

2.3 Proof Methods

The basic building blocks of Isabelle proofs are the proof methods which modify the goal state. If there are several goals in the goal state it depends on the specific method which goals are affected by it. In most cases only the first goal is affected.

2.3.1 The empty Method

The empty method is denoted by a single minus sign

```
-
```

If no input facts are passed to it, it does nothing, it does not alter the goal state.
The empty method is useful at the beginning of a structured proof of the form

proof *method* ST_1 ... ST_n **qed**

If the statements ST_1 ... ST_n shall process the unmodified original goal the empty method must be specified for *method*, thus the structured proof becomes

proof - ST_1 ... ST_n **qed**

Note that it is possible to syntactically omit the *method* completely, but then it defaults to the method named *standard* which alters the goal state (see below).
If input facts are passed to the empty method, it affects all goals by inserting the input facts as assumptions after the existing assumptions. If the input facts are $F_1,\ldots,F_m$ a goal of the form $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ is replaced by the goal $[\![A_1;\ldots;A_n;F_1,\ldots,F_m]\!] \Longrightarrow C$.
This makes it possible to transfer facts stored in the proof context to the goal state where they are stored as rule assumptions (see Section 2.2.2). Since this way of storing facts is not useful for structured proofs it is normally useless to input facts into a structured proof with the empty method as initial method.
If a goal statement instead uses a proof script as subproof the script can be started by applying the empty method to transfer input facts into the goal state for use by further method applications:

... **then** **have** "*C*" **apply** - MA_1 ... MA_n **done**

If the current facts are $F_1,\ldots,F_m$ before the **then** statement the goal state in the subproof contains the goal $[\![F_1,\ldots,F_m]\!] \Longrightarrow C$ before method application MA_1, so the facts are available for use in the proof script.

2.3.2 Terminating Proof Scripts

As described in Section 2.2.1 a proof is complete when its goal state is empty. In a structured proof goals are removed from the goal state by successful **show** statements. Therefore a structured proof is usually terminated by a **show** statement which removes the last goal in the goal state.

Proof scripts, instead, are intended to store facts as rule assumptions in the goal state (see Section 2.2.3). Then the proof of a goal is complete when the conclusion of the current goal unifies with one of its assumptions (see Section 2.2.2).

The *assumption* Method

Such goals are removed from the goal state by the method

assumption

The method only affects the first goal. If that goal has the form $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ and one assumption A_i unifies with C the method removes the goal, otherwise an error is signaled.

The *assumption* method is automatically applied by a proof part of the form

qed *method*

after applying the specified *method*. The application of the *assumption* method is repeated as long as it is applicable, thus **qed** removes all goals from the goal state where the conclusion matches an assumption. The same holds for the abbreviated forms **by** *method* and **by** $method_1$ $method_2$ (see Section 2.2.3).

Therefore a proof script consisting of method applications MA_1 ... MA_n can be terminated by **by** - if the method applications refine all goals to the form where the conclusion unifies with an assumption. Note that **done** does not remove such goals, when it is used to terminate a proof it expects that the goal state is already empty.

2.3.3 Basic Rule Application

As described in Section 2.2.2 the basic step in the construction of a proof is to establish the connection between a fact F_i and a fact F_{i+1} in the fact sequence. Assume that there is already a valid derivation rule $RA_i \Longrightarrow RC_i$ named r_i where RA_i unifies with F_i and RC_i unifies with F_{i+1}. Then the connection can be established by applying that rule.

The *rule* Method

A rule is applied by the method

```
rule name
```

where *name* is the name of a valid rule. The method only affects the first goal. If that goal has the form $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ and the rule referred by *name* has the form $[\![RA_1;\ldots;RA_m]\!] \Longrightarrow RC$ the method first unifies RC with the goal conclusion C. That yields the specialized rule $[\![RA_1';\ldots;RA_m']\!] \Longrightarrow RC'$ where RC' is syntactically equal to C and every RA_j' results from RA_j by substituting unknowns by the same terms as in RC'. If the goal does not contain unknowns (which is the normal case) C is not modified by the unification. If the unification fails the method cannot be executed on the goal state and an error is signaled. Otherwise the method replaces the goal by the m new goals $[\![A_1;\ldots;A_n]\!] \Longrightarrow RA_j'$.

If the rule has the form $RA \Longrightarrow RC$ with only one assumption the method replaces the goal by the single new goal $[\![A_1;\ldots;A_n]\!] \Longrightarrow RA'$. If the rule is a formula RC without any assumptions the method removes the goal without introducing a new goal.

Note that if facts are stored as rule assumptions in the goal state (see Section 2.2.2) an application of method *rule* preserves these facts and copies them to every new goal.

If an assumption RA_j is again a rule (i.e., the applied rule is a meta-rule) and has the form $[\![B_1;\ldots;B_k]\!] \Longrightarrow B$ the jth new goal becomes $[\![A_1;\ldots;A_n]\!] \Longrightarrow ([\![B_1';\ldots;B_k']\!] \Longrightarrow B')$ which by definition of the $[\![\ldots;\ldots]\!]$ syntax (see Section 2.1.6) is equivalent to $[\![A_1;\ldots;A_n;B_1';\ldots;B_k']\!] \Longrightarrow B')$. In this way the rule can introduce additional assumptions in the resulting goals, which are inserted after the existing assumptions.

Using the *rule* Method for Backward Reasoning Steps

Assume that during a proof for $A \Longrightarrow C$ as described in Section 2.2.2 the intermediate fact sequence F_{i+1}, $\ldots$ F_{n-1}, C has already been constructed by backward reasoning, i.e., the current goal is F_{i+1}. If r_i names a rule $RA_i \Longrightarrow RC_i$, the successful method application

apply *(rule r_i)*

will replace the current goal by F_i and thus extend the fact sequence to F_i, $\ldots$, F_{n-1}, C. The fact F_i is the specialized assumption RA_i' constructed by the method from the assumption RA_i of rule r_i. Together, this application of the *rule* method implements the backwards reasoning step from F_{i+1} to F_i.

Therefore the fact sequence F_1, ..., F_{n-1}, C of the complete proof can be constructed by the proof script consisting of the backward reasoning steps

apply (*rule* r_{n-1})
...
apply(*rule* r_1)
apply(*assumption*)
done

Note that we used the *assumption* method to remove the goal $A \Longrightarrow F_1$ by unifying F_1 with A. Alternatively we can use the form

apply (*rule* r_{n-1})
...
apply(*rule* r_1)
by -

where the *assumption* method is applied implicitly, as described in Section 2.3.2, or even shorter

apply (*rule* r_{n-1})
...
by (*rule* r_1)

If the example from Section 2.2.2 is proved this way the theorem is written together with its proof as

theorem "(x::nat) < 5 $\Longrightarrow$ 2*x+3 $\le$ 2*5 + 3"
  apply (*rule example2*)
  by (*rule example1*)

Note that the assumption A of the initial goal must be reached exactly by the sequence of rule applications. If it is replaced in the example by the stronger assumption x < 3 the rule applications will lead to the goal x < 3 $\Longrightarrow$ x < 5 which is trivial for the human reader but not applicable to the *assumption* method.

Automatic Rule Selection

The *rule* method can be specified in the form

rule

without naming the rule to be applied. Then it selects a rule automatically. It uses the first rule from the internal fact set *intro* for which the conclusion unifies with the goal conclusion. If there is no such rule in the set an error is signaled.

If the rules *example1* and *example2* would be in the *intro* set, the example proof could be written as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
  apply rule
  by rule
```

Introduction Rules

The set *intro* is intended by Isabelle for a specific kind of rules called "introduction rules". In such a rule a specific function *f* (perhaps written using an operator name) occurs only in the conclusion, but not in any assumption, hence it is "introduced" by the rule.

When an introduction rule for *f* is applied to a goal by the *rule* method, it works "backwards" and removes *f* from the goal. If the set *intro* only contains introduction rules and no rule adds a function which has been removed by another rule in the set, an iterated application of the *rule* method with automatic rule selection will "deconstruct" the goal: every step removes a function from the goal. The iteration stops when no rule in *intro* is applicable. In some sense the resulting goals are simpler because the set of functions used in them has been reduced. Some proofs can be written using this technique, however, they depend on the careful selection of the introduction rule in *intro*.

This proof technique can also be applied by the method

intro $name_1$ ... $name_n$

where $name_1$ ... $name_n$ refer to valid rules. It iterates applying rules from the given set to a goal in the goal state as long as this is possible. It is intended to be used with introduction rules. Then it automatically deconstructs the goals as much as possible with the given rules. Note that the method does not use the *intro* set.

A rule is also called an introduction rule for a function *f* if *f* occurs in some assumption(s) but in a different form (usually applied to other arguments). Then an application by the *rule* method will replace a term containing *f* by a different term containing *f*. The idea is to replace terms in several steps using one or more introduction rules until finally removing *f* completely.

The rule *example1* from Section 2.1.7 is an introduction rule for both functions *(≤)* and *(*)*, but it is only applicable to special uses of them and it replaces them by the function *(<)* which also is only useful for some specific proofs. In the rule *example2* the function *(≤)* also occurs in the assumption, however applied to other arguments, therefore it can also be considered as an introduction rule for *(≤)*.

Using the *intro* method the example proof can be written as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
  by (intro example1 example2)
```

The rules usually present in *intro* in Isabelle are carefully selected to be as generally applicable and useful for a large number of proofs as possible. Besides by the *rule* method the *intro* set is also used internally by some of the automatic methods described in Section 2.3.7.

If the cursor in the text area is positioned in a proof, introduction rules applicable to the first goal in the goal state of the enclosing proof can be searched using the keyword *intro* as criterion for a search by **find_theorems** or in the Query panel, as described in Section 2.1.7. It finds all named facts which can be applied by the *rule* method to the first goal, i.e., the conclusions can be unified.

The *standard* Method

The method

```
standard
```

is a method alias which can be varied for different Isabelle applications. Usually it is mainly an alias for the *rule* method.

The *standard* method is the default, if no method is specified as the initial step in a structured proof. Thus

proof ST_1 ... ST_n **qed**

is an abbreviation for

proof *standard* ST_1 ... ST_n **qed**

Note that the *standard* method as initial method in a structured proof will usually affect the goal by applying a rule from the set *intro* to it. That may be useful in some cases, but it has to be taken into account when writing the statements of the proof. If the rules *example1* and *example2* are again considered to be in the set *intro*, the example proof could be written as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
proof
  show "x < 5 ⟹ 2*x ≤ 2*5" by (rule example1)
qed
```

because *proof* (without the empty method -) already applies the rule *example2* by applying the *standard* method. It replaces the original goal by $x < 5 \Longrightarrow 2*x \leq 2*5$, so only this goal remains to be proved. However, this goal replacement is often not apparent to the reader of the proof. Therefore this form of structured proof should be used with care, it is only intended for some standard cases where the goal replacement is clearly expected by the proof reader and writer. Also note that *?thesis* still abbreviates the original goal conclusion and thus cannot be used in the proof anymore.

In the abbreviated form **by** *method* of a structured proof the method cannot be omitted, but the proof **by** *standard* can be abbreviated to

..

(two dots). It can be used as complete proof for a proposition which can be proved by a single automatic rule application. Since in the example proof also rule *example1* could be automatically selected by the *standard* method, it could be further abbreviated as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
proof
  show "x < 5 ⟹ 2*x ≤ 2*5" ..
qed
```

2.3.4 Rule Application in Forward Reasoning

Assume that during a proof for $A \Longrightarrow C$ as described in Section 2.2.2 the intermediate fact sequence $A,\ F_2,\ \ldots,\ F_i$ has already been constructed by forward reasoning and has been stored in the proof context. Then the next step is to state fact F_{i+1} and prove it using a rule $RA_i \Longrightarrow RC_i$ named r_i.

Using the *rule* Method for Forward Reasoning Steps

Using method *rule* the step can be started by the statement

have "F_{i+1}" **proof** ($rule\ r_i$)

The goal of this subproof is simply F_{i+1}, so applying the *rule* method with r_i will result in the new goal RA_i' which unifies with F_i. The subproof is not finished, since its goal state is not empty. But the goal unifies with an already known fact. The proof method

fact name

can be used to remove that goal. The method only affects the first goal. If the fact referred by *name* unifies with it, the goal is removed, otherwise an error is signaled.

Using this method the forward reasoning step can be completed as

```
have "$F_{i+1}$" proof (rule $r_i$) qed (fact $f_i$)
```

if F_i has been named f_i. This can be abbreviated (see Section 2.2.3) to

```
have "$F_{i+1}$" by (rule $r_i$) (fact $f_i$)
```

Therefore the fact sequence A, F_2, ..., F_{n-1}, C of the complete proof for the goal $A \Longrightarrow C$ can be constructed by the structured proof of the form

```
proof -
assume a: "A"
have $f_2$: "$F_2$" by (rule $r_1$) (fact a)
...
have $f_{n-1}$: "$F_{n-1}$" by (rule $r_{n-2}$) (fact $f_{n-2}$)
show "?thesis" by (rule $r_{n-1}$) (fact $f_{n-1}$)
qed
```

where *?thesis* abbreviates the conclusion *C*, as usual.

If the example from Section 2.2.2 is proved this way the theorem is written together with its proof as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
proof -
  assume a: "x < 5"
  have $f_2$: "2*x ≤ 2*5" by (rule example1) (fact a)
  show ?thesis by (rule example2) (fact $f_2$)
qed
```

The *fact* method can be specified in the form

```
fact
```

without naming the fact to be used. Then it selects a fact automatically. It uses the first fact from the proof context which unifies with the goal. If there is no such fact in the proof context an error is signaled.

Thus the example can be written without naming the facts as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
proof -
  assume "x < 5"
  have "2*x ≤ 2*5" by (rule example1) fact
  show ?thesis by (rule example2) fact
qed
```

Input Facts for the *rule* Method

If input facts F_1, ..., F_n are passed to the *rule* method, they are used to remove assumptions from the rule applied by the method. If the rule has the form $[\![RA_1;\ldots;RA_{n+m}]\!] \Longrightarrow RC$ and every fact F_i unifies with assumption RA_i the first n assumptions are removed and the rule becomes $[\![RA_{n+1}';\ldots;RA_{n+m}']\!] \Longrightarrow RC'$, specialized according to the term substitutions performed by the unifications. Then it is applied to the first goal in the usual way. If there are more input facts than assumptions or if a fact does not unify, an error is signaled.

This behavior of the *rule* method can be explained as follows: as described above, for every assumption in the applied rule the method creates a goal which has the assumption as conclusion. As usual, the goal is considered solved, if the conclusion unifies with a fact in the proof context. By unifying the input facts with the rule assumptions the method determines the goals which would immediately be solved and thus can be omitted, then it removes the assumptions from the rule so that the corresponding goals are never created.

This allows to establish the connection from a fact F_i to F_{i+1} in a fact chain by a forward reasoning step of the form

from f_i **have** "F_{i+1}" **by** (rule r_i)

where f_i names the fact F_i. When it is input to the goal statement it is passed to the *rule* method and removes the assumption from the applied rule $RA_i \Longrightarrow RC_i$, resulting in the assumption-less "rule" RC_i. When it is applied to the goal F_{i+1} it unifies and removes the goal, thus the subproof is complete.

For the fact sequence chaining can be used to write a structured proof without naming the facts:

proof -
assume "F_1"
then have "F_2" **by** (rule r_1)
...
then have "F_{n-1}" **by** (rule r_{n-2})
then show "F_n" **by** (rule r_{n-1})
qed

As described in Section 2.3.3 the subproof by (rule r_i) can be abbreviated as `..` if the rule r_i is in the *intro* rule set.

If the example from Section 2.2.2 is proved this way the theorem is written together with its proof as

theorem "$(x::nat) < 5 \Longrightarrow 2*x+3 \leq 2*5 + 3$"
proof -

```
  assume "x < 5"
  then have "2*x ≤ 2*5" by (rule example1)
  then show ?thesis by (rule example2)
qed
```

The Method *this*

Rule application can also be done by the method

```
this
```

Instead of applying a named method, it applies the input fact as rule to the first goal.

If several input facts are given, the method applies them exactly in the given order. Therefore the fact sequence can also be constructed by a structured proof of the form:

```
proof -
assume "F_1"
with r_1 have "F_2" by this
...
with r_{n-2} have "F_{n-1}" by this
with r_{n-1} show "F_n" by this
qed
```

The **with** statement inserts the explicitly specified facts *before* the current facts. Therefore every goal statement for F_i gets as input the rule r_{i-1} followed by the chained fact F_{i-1}. The method *this* first applies the rule which replaces the goal by F_{i-1}. Then it applies the fact F_{i-1} as rule to this goal which removes it and finishes the subproof.

The proof

```
by this
```

can be abbreviated by . (a single dot).

Therefore the example from Section 2.2.2 can also be proved in the form

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
proof -
  assume "x < 5"
  with example1 have "2*x ≤ 2*5" .
  with example2 show ?thesis .
qed
```

The Methods *frule* and *drule*

Instead of storing facts in the proof context and using a structured proof for a forward reasoning proof the facts may be stored as rule assumptions in the goal state and the forward reasoning proof may be written as a proof script (see Section 2.2.3).

To construct a proof for the goal $A \Longrightarrow C$ as fact sequence $A,\ F_2\ \ldots\ F_n$ by forward reasoning as described in Section 2.2.2 in this way, the intermediate fact sequence $A,\ F_2,\ \ldots,\ F_i$ is stored in the extended goal $[\![A;\ F_2;\ \ldots;\ F_i]\!] \Longrightarrow C$ where the last assumption is the current fact F_i. Then the next forward reasoning step consists of adding the fact F_{i+1} as new assumption to the goal and prove it using a rule $RA_i \Longrightarrow RC_i$ named r_i. When the fact sequence is complete the goal is $[\![A;\ F_2;\ \ldots;\ F_n]\!] \Longrightarrow C$ and F_n unifies with C, thus an application of method `assumption` will remove the goal and terminate the proof (see Section 2.3.2).

A rule is applied for forward reasoning by the method

```
frule name
```

where `name` is the name of a valid rule. The method only affects the first goal. If that goal has the form $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ and the rule referred by `name` has the form $[\![RA_1;\ldots;RA_m]\!] \Longrightarrow RC$ the method first unifies the first assumption RA_1 in the rule with the first assumption A_k of the $A_1\ \ldots\ A_n$ which can be unified with RA_1. That yields the specialized rule $[\![RA_1';\ldots;RA_m']\!] \Longrightarrow RC'$ where RA_1' is syntactically equal to A_k and every RA_j' with $j > 1$ and RC' results from RA_j or RC, respectively, by substituting unknowns by the same terms as in RA_1'. If the goal does not contain unknowns (which is the normal case) A_k is not modified by the unification. If no A_i unifies with RA_1 the method cannot be executed on the goal state and an error is signaled. Otherwise the method replaces the goal by the $m-1$ new goals $[\![A_1;\ldots;A_n]\!] \Longrightarrow RA_j'$ for $j > 1$ and the goal $[\![A_1;\ldots;A_n;RC']\!] \Longrightarrow C$.

If the rule has the form $RA \Longrightarrow RC$ with only one assumption the method replaces the goal by the single new goal $[\![A_1;\ldots;A_n;RC']\!] \Longrightarrow C$, i.e., it adds the conclusion of the rule as assumption in the goal. If the rule is a formula RC without any assumptions the method is not applicable and signals an error.

Since the first assumption RA_1 in the rule plays a special role in this context, it is also called the “major premise” of the rule.

Together, a forward reasoning step as described above can be implemented by the method application

apply *(frule r_i)*

and the full proof script for a forward reasoning proof has the form

apply(*frule* r_1)
...
apply (*frule* r_{n-1})
apply(*assumption*)
done

or shorter

apply (*frule* r_1)
...
by (*frule* r_{n-1})

If the example from Section 2.2.2 is proved this way the theorem is written together with its proof as

theorem *"(x::nat) < 5 $\Longrightarrow$ 2*x+3 $\leq$ 2*5 + 3"*
  apply (*frule example1*)
  by (*frule example2*)

However, this form of forward reasoning proof has several drawbacks. First, as always in proof scripts, the facts F_i are not specified explicitly, they are constructed implicitly by the *frule* method and can only be seen by interactively inspecting the goal state. Next, since the current fact is the last assumption in the goal, it is not guaranteed that the rule r_i is applied to it. If a previous assumption also unifies with the major premise of r_i the rule is not applied in the intended way. Finally, it is not possible to generalize this approach to proofs with several branches. The branches cannot be joined, because *frule* always takes only one assumption into account.

The second drawback can be compensated for by using another method for applying the rule. This is done by the method

drule name

where *name* is the name of a valid rule. The method works like *frule*, but instead of adding *RC'* to the assumptions it replaces A_k by it. Thus the method replaces the goal by the *m-1* new goals $[\![A_1;\ldots;A_{k-1};\ A_{k+1};\ldots;A_n]\!] \Longrightarrow RA_j'$ for $j > 1$ and the goal $[\![A_1;\ldots;A_{k-1};A_{k+1};\ldots;A_n;RC']\!] \Longrightarrow C$.

When using *drule* for constructing a proof in the way described above, it always replaces the current fact by the next one in the fact sequence. The intermediate fact sequence is represented by the goal $F_i \Longrightarrow C$. Since the current fact is the only assumption present in the goal, the step **apply** (*drule* r_i) is always applied to it and replaces the goal by $F_{i+1} \Longrightarrow C$, as intended.

The methods *frule* and *drule* do not support input facts.

Destruction Rules

Not all rules can always usefully be applied by `frule` and `drule`. Since both methods only unify their first assumption (the major premise) of the rule with a term in the goal and then replace it by the conclusion, the first assumption should have some effect on the conclusion. In particular, the conclusion should not be a single unknown which does not occur in the first assumption.

If additionally a specific function `f` (perhaps written using an operator name) occurs only in the first assumption and neither in the conclusion, nor in other assumptions, the rule is called a "destruction rule" for `f`. If it is applied in forward direction, such as with `frule` and `drule`, `f` will be removed from the goal, it will be "destructed".

Similar to introduction rules (see Section 2.3.3) `f` may occur in the conclusion if it has a different form, so that it may be removed by several steps through intermediate forms.

Analogous to the `intro` set for introduction rules there is an internal fact set `dest` for destruction rules. It is used by some automatic methods, however, it is not used for automatically selecting rules for `frule` and `drule`.

If the cursor in the text area is positioned in a proof, destruction rules applicable to the first goal in the goal state of the enclosing proof can be searched using the keyword `dest` as criterion for a search by **find_theorems** or in the Query panel, as described in Section 2.1.7. It finds all named facts which can be applied by the `frule` or `drule` method to the first goal, i.e., the major premise unifies with a goal assumption.

The rule `example1` from Section 2.1.7 is a destruction rule for function `(<)`, but it is also only applicable to special uses of it and it replaces it by the functions `(≤)`, `(*)`, and `(+)` which does not help for most proofs. In the rule `example2` the operator `(≤)` also occurs in the conclusion, but in different form. Therefore it can be considered as destruction rule for `(≤)`, although the form in the conclusion is more complex which also does not help for most proofs.

2.3.5 Composed Proof Methods

Proof methods can be composed from simpler methods with the help of "method expressions". A method expression has one of the following forms:

- m_1`, ..., `m_n : a sequence of methods which are applied in their order,
- m_1`; ...; `m_n : a sequence of methods where each is applied to the goals created by the previous method,

- m_1| ...| m_n : a sequence of methods where only the first applicable method is applied,
- `m[n]` : the method `m` is applied to the first `n` goals,
- `m?` : the method `m` is applied if it is applicable,
- `m+` : the method `m` is applied once and then repeated as long as it is applicable.

Parentheses are used to structure and nest composed methods.

Composed methods can be used to combine method applications to a single step. Using composed methods the example backward reasoning proof script from Section 2.3.3 can be written as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
  apply(rule example2,rule example1,assumption)
  done
```

In particular, it is also possible to apply an arbitrarily complex composed method as initial method in a structured proof. Using composed methods the first example forward reasoning proof in Section 2.3.4 can be written as

```
theorem "(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"
proof -
  assume a: "x < 5"
  have f2: "2*x ≤ 2*5" by (rule example1,fact a)
  show ?thesis by (rule example2,fact f2)
qed
```

2.3.6 The Simplifier

A common proof technique is “rewriting”. If it is known that a term `a` is equal to a term `b`, some occurrences of `a` in a proposition can be replaced by `b` without changing the validity of the proposition.

Equality of two terms `a` and `b` can be expressed by the proposition `a = b`. If that proposition has been proved to be valid, i.e., is a fact, `a` can be substituted by `b` and vice versa in goals during a proof.

The *subst* Method

Rewriting is performed by the method

```
subst name
```

where *name* references an equality fact. The method only affects the first goal. If the referenced fact has the form `a = b` the method replaces the first occurrence of `a` in the goal conclusion by `b`. The order of the terms in the equality fact matters, the method always substitutes the term on the left by that on the right.

If the equality contains unknowns unification is used: `a` is unified with every sub-term of the goal conclusion, the first match is replaced by `b'` which is `b` after substituting unknowns in the same way as in `a`. If there is no match of `a` in the goal conclusion an error is signaled.

For a goal $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$ the method only rewrites in the conclusion C. The first match in the assumptions $A_1 \ldots A_n$ can be substituted by the form

```
subst (asm) name
```

If not only the first match shall be substituted, a number of the match or a range of numbers may be specified in both forms as in

```
subst (asm) (i..j) name
```

The equality fact can also be a meta equality of the form $a \equiv b$. Therefore the method can be used to expand constant definitions. After the definition

definition *"inc x ≡ x + 1"*

the method *subst inc_def* will rewrite the first occurrence of a function application *(inc t)* in the goal conclusion to *(t + 1)*. Remember from Section 2.1.4 that the defining equation is automatically named *inc_def*. Note the use of unification to handle the actual argument term *t*.

The equality fact may be conditional, i.e., it may be a derivation rule with assumptions of the form $[\![RA_1; \ldots; RA_m]\!] \Longrightarrow a = b$. When the *subst* method applies a conditional equation of this form to a goal $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$, it adds the goals $[\![A_1; \ldots; A_n]\!] \Longrightarrow RA_i'$ to the goal state after rewriting, where RA_i' result from RA_i by the unification of a in C. These goals are inserted before the original goal, so the next method application will usually process the goal $[\![A_1; \ldots; A_n]\!] \Longrightarrow RA_1'$.

As an example if there are theorems

theorem *eq1: "n = 10 ⟹ n+3 = 13"* **for** *n::nat* ⟨*proof*⟩
theorem *eq2: "n = 5 ⟹ 2*n = 10"* **for** *n::nat* ⟨*proof*⟩

the method *subst (2) eq2* replaces the goal *(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3* by the goals

```
x < 5 ⟹ 5 = 5
x < 5 ⟹ 2 * x + 3 ≤ 10 + 3
```

where the first is trivial (but still must be removed by applying a rule). The second goal is replaced by the method `subst (2) eq1` by

```
x < 5 ⟹ 10 = 10
x < 5 ⟹ 2 * x + 3 ≤ 13
```

Note that the method `subst eq2` would unify `2*n` with the first match `2*x` in the original goal and replace it by

```
x < 5 ⟹ x = 5
x < 5 ⟹ 10 + 3 ≤ 2 * 5 + 3
```

where the first goal cannot be proved because it is invalid.

Simplification

If the term `b` in an equation `a = b` is in some sense "simpler" than `a`, the goal will also become simpler by successful rewriting with the equation. If there are several such equations a goal can be replaced by successively simpler goals by rewriting with these equations. This technique can contribute to the goal's proof and is called "simplification".

Basically, simplification uses a set of equations and searches an equation in the set where the left hand side unifies with a sub-term in the goal, then substitutes it. This step is repeated until no sub-term in the goal unifies with a left hand side in an equation in the set.

It is apparent that great care must be taken when populating the set of equations, otherwise simplification may not terminate. If two equations `a = b` and `b = a` are in the set simplification will exchange matching terms forever. If an equation `a = a+0` is in the set, a term matching `a` will be replaced by an ever growing sum with zeroes.

Simplification with a set of definitional equations from constant definitions (see Section 2.1.4) always terminates. Since constant definitions cannot be recursive, every substitution removes one occurrence of a defined constant from the goal. Simplification terminates if no defined constant from the set remains in the goal. Although the resulting goal usually is larger than the original goal, it is simpler in the sense that it uses fewer defined constants.

If the set contains conditional equations, simplification may produce additional goals. Then simplification is applied to these goals as well. Together, simplification may turn a single complex goal into a large number of simple goals, but it cannot reduce the number of goals. Therefore simplification is usually complemented by methods which remove trivial goals like `x = x`, `A ⟹ A`, and `True`. Such an extended simplification may completely solve and remove the goal to which it is applied.

The *simp* Method

Isabelle supports simplification by the method

```
simp
```

which is also called “the simplifier”. It uses the dynamic fact set *simp* as the set of equations, which is also called “the simpset”. The method only affects the first goal. If no equation in the simpset is applicable to it or it is not modified by the applicable equations an error is signaled.

The *simp* method simplifies the whole goal, i.e., it applies rewriting to the conclusion and to all assumptions.

The simpset may contain facts which are not directly equations, but can be converted to an equation. In particular, an arbitrary derivation rule $\llbracket A_1;\ \ldots;\ A_n\rrbracket \Longrightarrow C$ can always be converted to the conditional equation $\llbracket A_1;\ \ldots;\ A_n\rrbracket \Longrightarrow C = True$. The simplifier (and also the *subst* method) performs this conversion if no other conversion technique applies, therefore the simpset may actually contain arbitrary facts.

The *simp* method also detects several forms of trivial goals and removes them. Thus a complete proof may be performed by a single application of the simplifier in the form

by *simp*

In Isabelle HOL (see Section 3) the simpset is populated with a large number of facts which make the simplifier a very useful proof tool. Actually all examples of facts used in the previous sections can be proved by the simplifier:

theorem *example1: "(x::nat) < c* $\Longrightarrow$ *n*x* $\le$ *n*c"* **by** *simp*
theorem *example2: "(x::nat)* $\le$ *c* $\Longrightarrow$ *x + m* $\le$ *c + m"* **by** *simp*
theorem *"(x::nat) < 5* $\Longrightarrow$ *2*x+3* $\le$ *2*5 + 3"* **by** *simp*
theorem *eq1: "n = 10* $\Longrightarrow$ *n+3 = 13"* **for** *n::nat* **by** *simp*
theorem *eq2: "n = 5* $\Longrightarrow$ *2*n = 10"* **for** *n::nat* **by** *simp*

Configuring the Simplifier

The simplifier can be configured by modifying the equations it uses. The form

simp add: $name_1 \ldots name_n$

uses the facts $name_1, \ldots, name_n$ in addition to the facts in the simpset for its rewriting steps. The form

simp del: $name_1 \ldots name_n$

uses only the facts from the simpset without the facts *name*$_1$*, ..., name*$_n$, and the form

simp only: name$_1$ *... name*$_n$

uses only the facts *name*$_1$*, ..., name*$_n$. The three forms can be arbitrarily combined.

As usual, a theorem may be added permanently to the simpset as described in Section 2.1.7 by specifying it as

theorem *[simp]: "prop"* ⟨*proof*⟩

and the defining equation of a definition can be added by

definition *name::type* **where** *[simp]: "name ≡ term"*

Adding own constant definitions to the simplifier is a common technique to expand the definition during simplification. However, this may also have a negative effect: If an equation has been specified using the defined constant, it is no more applicable for rewriting after expanding the definition. Note that the facts in the simpset and the facts provided by *add:*, *del:*, and *only:* are not simplified themselves, the defined constant will not be expanded there.

Therefore it is usually not recommended to add defining equations to the simpset permanently. Instead, they can be specified by *add:* when they really shall be expanded during simplification.

Splitting Terms

There are certain terms in which the simplifier will not apply its simpset rules. A typical example are terms with an internal case distinction (see Section 4.1.3). To process such terms in a goal conclusion the terms must be split. Splitting a term usually results in several new goals with simpler terms which are then further processed by the simplifier.

Term splitting is done by applying specific rules to the goal. These rules are called "split rules". Usually split rules are not automatically determined and applied by the simplifier, this must be configured explicitly in the form

simp split: name$_1$ *... name*$_n$

where the *name*$_i$ are the names of the split rules to use. This configuration can be arbitrarily combined with the other simplifier configuration options.

The usual form of a split rule is a proposition

"?P(term) = ((Q$_1$ $\longrightarrow$ *?P(term*$_1$*))* $\wedge$ ... $\wedge$ *(Q*$_n$ $\longrightarrow$ *?P(term*$_n$*)))"*

where the `term`$_i$ are subterms of `term` and every `Q`$_i$ represents a condition for which `term` can be reduced to `term`$_i$. The simplifier applies such a split rule to a goal $[\![A_1;\ldots;A_m]\!] \Longrightarrow C$ by first unifying the left hand side with the conclusion `C` (which succeeds if `term` occurs in `C`), then replacing it by the conjunction on the right, then splitting the goal into a separate goal for every conjunct, and finally moving every `Q`$_i$ to the assumptions of their goal. Thus the resulting goals have the form

$[\![A_1;\ldots;A_m;Q_1]\!] \Longrightarrow C_1$
...
$[\![A_1;\ldots;A_m;Q_n]\!] \Longrightarrow C_n$

where C_i is constructed from `C` by mainly replacing `term` by `term`$_i$.

Note that this form of a split rule can only be applied for splitting terms in the conclusion of a goal. See the Isabelle documentation for other forms which split terms in assumptions of a goal.

Searching Simplifier Equations

Named facts applicable for simplification may be searched using the command **find_theorems** or in the Query panel, as described in Section 2.1.7, using a `criterion`$_i$ of the form `simp: termpat` where `termpat` is a term pattern (a term which may contain unknowns) specified in inner syntax.

Such a search finds named facts where the conclusion is an equation (using either the operator `=` or the meta equality $\equiv$) and the left side unifies with the specified term pattern. It also finds facts where the conclusion unifies with the term pattern, if the term pattern has type `bool`, because such facts are equivalent to an equation with `True` (see above). Facts are found independent of whether they are in the simpset or not.

A found fact may be used by specifying it for the `subst` method or, if not yet in the simpset, by configuring the `simp` method to use it with the help of `add:` or `only:`.

Input Facts for the `simp` Method

As usual, facts may be input to the `simp` method. Like the empty method (see Section 2.3.1) it inserts these facts as assumptions into the goal, before it starts simplification. Since simplification is also applied to the assumptions, the input facts will be simplified as well.

As a possible effect of this behavior, after simplifying an input fact and the goal conclusion the results may unify, leading to the situation where the goal is removed by the `assumption` method (see Section 2.3.2). This is also done by the simplifier, hence in this way the input fact may contribute to prove the goal.

The *simp_all* Method

The method

simp_all

behaves like the *simp* method but processes all goals. It inserts input facts to all goals in the goal state and simplifies them. If it fails for all goals an error is signaled. Otherwise it simplifies only the goals for which it does not fail. If it achieves to remove all goals the proof is finished.

The *simp_all* method can be configured by *add:*, *del:*, *only:*, and *split:* in the same way as the *simp* method.

The *simp_all* method is useful, if first a method *m* is applied to the goal which splits it into several subgoals which all can be solved by simplification. Then the complete proof can be written as

by *m simp_all*

Debugging the Simplifier

If the simplifier fails, it may be difficult to find out the reason. There are several debugging techniques which may help.

The content of the simpset can be displayed by the command

print_simpset

which may be specified in the proof text in modes *proof(prove)* and *proof(state)* and outside of proofs. In the interactive editor the result is displayed in the Output panel (see Section 1.2.3).

There is also a simplifier trace which displays the successful rewrite steps. It is activated by the command

declare *[[simp_trace_new depth=n]]*

outside a theorem or definition. The number *n* should be atleast *2*. When the cursor is positioned on an application of the *simp* method the button “Show trace” can be used in the Simplifier Trace panel to display the trace in a separate window. See the documentation for more information about how to use the trace.

Another technique is to replace the *simp* method by a sequence of *subst* method applications and explicitly specify the equations which should have been used. To do this for a structured proof, replace it by a proof script for the *subst* method applications.

2.3.7 Other Automatic Proof Methods

Isabelle provides several other proof methods which internally perform several steps, like the simplifier.

Automatic Methods

The following list contains automatic methods other than `simp`:

- `blast` mainly applies logical rules and can be used to solve complex logical formulas.
- `clarify` is similar but does not split goals and does not follow unsafe paths. It can be used to show the problem if `blast` fails.
- `auto` combines logical rule application with simplification. It processes all goals and leaves those it cannot solve.
- `clarsimp` combines `clarify` with simplification. It processes only the first goal and usually does not split goals.
- `fastforce` uses more techniques than `auto`, but processes only the first goal.
- `force` uses even more techniques and tries to solve the first goal.
- `linarith` solves linear arithmetic problems (on integer and real numbers) for the first goal. It is automatically included by the simplifier.
- `arith` uses more techniques than `linarith` but may be slower.

The methods which do simplification can be configured like the simplifier by adding specifications `simp add:`, `simp del:`, `simp only:`, and `split:`. For example, additional simplification rules can be specified for the `auto` method in the form

`auto simp add:` *name*$_1$ ... *name*$_n$

For more information about these methods see the Isabelle documentation.

Trying Methods

Instead of manually trying several automatic methods it is possible to specify the command

try

anywhere in mode `proof(prove)`, i.e. at the beginning of a proof or in a proof script. It will try many automatic proof methods and describe the result in the Output window. It may take some time until results are displayed, in particular, if the goal is invalid and cannot be proved.

If **try** finds a proof for one or more goals it displays it as a single (composed) proof method, which, by clicking on it can be copied to the cursor position in the text area. The **try** command must be removed manually.

If **try** tells you that the goal can be "directly solved" by some fact, you can usually prove it by the *fact* method, but that also means that there is already a fact of the same form and your theorem is redundant.

It may also be the case that **try** finds a counterexample, meaning that the goal is invalid and cannot be proved.

2.4 Case Based Proofs

If during a proof the goal state contains several goals they are often proved sequentially. Although there are proof methods which process several goals at once (see examples in Section 2.3.7) most methods only process the first goal. In a proof script, when a method has solved and removed the first goal, the next goal will become the first and will be processed by the next method application steps. In a structured proof (see Section 2.2.3) it is not so simple. To prove a goal which is in the goal state its bound variables and assumptions have to be inserted into the local proof context (by **fix** and **assume** statements) and its conclusion must be stated by a **show** statement and must be proved. Isabelle provides some support for simplifying these tasks for working on a sequence of goals.

2.4.1 Named Proof Contexts

Isabelle supports some methods which are able to create "cases" for goals. A case contains bound variables and assumptions from a goal and it may contain other context elements, such as names for assumptions or assumption groups and an abbreviation for the conclusion of a goal. The cases are named. Using these names a proof context can be populated with all content of a case in a single step.

Since a case contains context elements it can be seen as a named context which has been prepared by a method for later use, but has not been "activated" yet. Usually a named context is used to initialize a new nested proof context immediately after its beginning by inserting the content of the named context into the new context.

The case Statement

The content of a case may be inserted into the current proof context by the statement

case *name*

where *name* is the case name. It mainly has the effect of the sequence

fix x_1 ... x_k
let ?a_1 = t_1 **and** ... ?a_m = t_m
assume *name:* "A_1" ... "A_n"

where x_1 ... x_k are the local variables, ?a_1, ..., ?a_m are the term abbreviations, and A_1, ..., A_n are the facts in the named context of the case. The facts are inserted as assumptions and the set of these assumptions is named using the case name. Moreover, like the **assume** statement, the **case** statement makes the assumed facts current.

Instead of using the case name for naming the assumptions an explicit assumption name *aname* may be specified:

case *aname: name*

If defined in the case, other names for single assumptions or assumption groups may be automatically introduced.

The local variables x_1 ... x_k are fixed by the **case** statement but are hidden, they cannot be used in the subsequent proof text. If they should be used, explicit names must be specified for them in the form

case (*name* y_1 ... y_j)

Then the names y_1 ... y_j can be used to reference the fixed variables in the current proof context. If fewer names are specified only the first variables are named, if more names are specified than there are local variables in the case an error is signaled.

When methods create a named context for a goal they usually only define the term abbreviation `?case` for the conclusion of the goal.

Proof Structure with Cases

The usual proof structure using cases consists of an initial method which creates cases and of a sequence of nested contexts (see Section 2.2.3). At its beginning each context is initialized by a **case** statement naming one of the created cases, at its end it uses a **show** statement to remove the corresponding goal:

proof *method*
case $name_1$
...
show *?case* ⟨*proof*⟩
next
case $name_2$
...
show *?case* ⟨*proof*⟩
next
...
next
case $name_n$
...
show *?case* ⟨*proof*⟩
qed

Every **show** statement uses the local term abbreviation *?case* to refer to the conclusion of the corresponding goal.

To find out which cases have been introduced by a method application, the command

print_cases

can be used at arbitrary places in the following proof to display the cases in the Output panel.

In the interactive editor also the Query panel (see Section 1.2.7) can be used to display the cases available at the cursor position by selecting the tab "Print Context" and checking "cases".

Also in the interactive editor, when the cursor is positioned on **proof** *method* where the method supports cases, a skeleton of a proof using the specific cases provided by the method is displayed in the Output panel. By clicking on it it may be copied into the text area immediately after the method specification.

2.4.2 The *goal_cases* Method

The simplest method with support for cases is

goal_cases

Without modifying the goal state it creates a named case for every existing goal. Input facts are ignored.

For a goal $\bigwedge x_1 \ldots x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow C$ the created named context contains the local variables $x_1 \ldots x_m$, the facts $A_1, \ldots, A_n$, and the term abbreviation *?case* bound to C. If the goal contains variables which are not explicitly bound by $\bigwedge$ these variables are not added to the context.

The effect is that if no other variables are fixed and no other facts are assumed a statement **show** *?case* after the corresponding **case** statement will refine the goal and remove it from the goal state.

The cases are named by numbers starting with *1*. If other names should be used they may be specified as arguments to the method:

goal_cases $name_1$ ... $name_n$

If fewer names are specified than goals are present, only for the first *n* goals cases are created. If more names are specified an error is signaled.

When *goal_cases* is used in a composed proof method it can provide cases for the goals produced by arbitrary other methods:

proof *(method, goal_cases)*

provides cases for all goals existing after *method* has been applied. If *method* does not split the goal there will be only one case. This can be useful to work with a goal produced by *method*. In particular, the conclusion of that goal is available as *?case*.

Note that the proof state(s) resulting from *goal_cases* are not visible for the reader of the proof. Therefore it should only be applied if the goals produced by *method* are apparent. In a case the goal conclusion can be shown partially or fully by defining a possibly abbreviated form of it by

let *"pattern" = ?case*

where the *pattern* may contain unknowns which abbreviate sub terms of the conclusion.

A better way to use cases is together with a method which combines both: creating new goals in a simple and expected way, and immediately creating cases only for these goals.

2.4.3 Case Based Reasoning

Case based proofs are especially convenient for implementing case based reasoning. The technique of "case based reasoning" uses a specific kind of forward reasoning steps (see Section 2.2.2). It adds a new fact F_{i+1} to the proof context "out of the blue" without proving it from the existing facts. For the proof to stay correct, this must be done for "all possible cases" of F_{i+1}, and the proof must be completed separately for each of them.

In its simplest form this can be done by adding an arbitrary fact F_{i+1} and its negation $\neg F_{i+1}$, this covers all possibilities, since F_{i+1} may be either *True* or *False*.

Consider the derivation rule `(x::nat) < c` $\Longrightarrow$ `n*x` $\le$ `n*c` named `example1` in Section 2.1.6. To prove it using case based reasoning the additional assumption that `n` is zero can be used. Then there are the two cases `n = 0` and `n` $\neq$ `0` and clearly these cover all possibilities. Using the first case as assumption implies that `n*x` and `n*c` are both zero and thus `n*x = n*c`. Using the second case as assumption together with the original assumption implies that `n*x < n*c`. Together the conclusion `n*x` $\le$ `n*c` follows.

Since the proof must be completed separately for every such case, a separate goal is required for each of them. This cannot be achieved by statements, the old goal must be replaced by several new goals which is only possible by applying a proof method.

More specific, if $Q_1 \ldots Q_k$ are propositions for the different cases which together cover all possibilities, a goal $\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow C$ must be replaced by the k goals

$\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n;\ Q_1]\!] \Longrightarrow C$
...
$\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n;\ Q_k]\!] \Longrightarrow C$

where every goal has one of the propositions Q_i as additional assumption.

Note that this technique extends the proof procedure described in Section 2.2.2. There a proof consisted of a single tree of facts which started at the assumptions and all branches joined to finally reach the conclusion. With case based reasoning at any position the remaining tree may be split into several "copies" with additional assumptions which then must all be completed separately.

Case Rules

This splitting of a goal into goals for different cases can be done by applying a single meta-rule of the specific form

$[\![Q_1 \Longrightarrow ?P;\ \ldots;\ Q_k \Longrightarrow ?P]\!] \Longrightarrow ?P$

where $Q_1 \ldots Q_k$ are all cases of the additional assumption. Such rules are called "case rules".

When this case rule is applied to the goal $\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow C$ as described in Section 2.3.3, it unifies $?P$ with the conclusion C and replaces the goal in the way described above.

A case rule is only valid, if the Q_i together cover all possibilities, i.e., if $Q_1 \vee \ldots \vee Q_k$ holds. If this has been proved the case rule is available as a fact which can be applied. Since the whole conclusion is the single unknown $?P$ it unifies with every proposition used as conclusion in a goal, hence a case rule can always be applied to arbitrary goals. It depends on the Q_i whether splitting a specific goal with the case rule is useful for proving the goal.

A case rule for testing a natural number for being zero would be

$\llbracket$`?n = 0` $\Longrightarrow$ `?P; ?n` $\neq$ `0` $\Longrightarrow$ `?P`$\rrbracket$ $\Longrightarrow$ `?P`

It contains the number to be tested as the unknown `?n`, so that an arbitrary term can be substituted for it. This is not automatically done by unifying `?P` with the goal's conclusion, thus the rule must be "prepared" for application to a specific goal. To apply it to the goal `(x::nat) < c` $\Longrightarrow$ `n*x` $\leq$ `n*c` in the intended way the unknown `?n` must be substituted by the variable `n` from the goal conclusion. If the prepared rule is then applied to the goal it splits it into the goals

$\llbracket$`(x::nat) < c; n = 0`$\rrbracket$ $\Longrightarrow$ `n*x` $\leq$ `n*c`
$\llbracket$`(x::nat) < c; n` $\neq$ `0`$\rrbracket$ $\Longrightarrow$ `n*x` $\leq$ `n*c`

which can now be proved separately.
Actually, the much more general case rule

$\llbracket$`?Q` $\Longrightarrow$ `?P;` $\neg$ `?Q` $\Longrightarrow$ `?P`$\rrbracket$ $\Longrightarrow$ `?P`

is used for this purpose. Here the unknown `?Q` represents the complete proposition to be used as additional assumption, therefore the rule can be used for arbitrary propositions. By substituting the term `n = 0` for `?Q` the rule is prepared to be applied in the same way as above.
Case rules may even be more general than shown above. Instead of a single proposition Q_i every case may have locally bound variables and an arbitrary number of assumptions, resulting in a meta-rule of the form

$\llbracket P_1; \ldots; P_k\rrbracket \Longrightarrow$ `?P`

where every P_i is a rule of the form

$\bigwedge x_{i1}\ldots x_{ipi}.\ \llbracket Q_{i1};\ldots;Q_{iqi}\rrbracket \Longrightarrow$ `?P`

That means, the P_i may be arbitrary rules but they must all have as conclusion the unknown `?P` which is also the conclusion of the overall case rule. When such a case rule is applied to a goal it splits the goal into `k` cases and adds the variables $x_{i1}\ \ldots\ x_{ipi}$ and the assumptions $Q_{i1}\ \ldots\ Q_{iqi}$ to the `i`th case.
Remember that to write your own case rule you have to specify a theorem which uses variables in place of the unknowns, such as

theorem `mycaserule: "`$\llbracket$`n = 0` $\Longrightarrow$ `P; n` $\neq$ `0` $\Longrightarrow$ `P`$\rrbracket$ $\Longrightarrow$ `P"` ⟨*proof*⟩

which will be converted to unknowns upon turning the proposition to a fact after the proof.

The *cases* Method

Case based reasoning can be performed in a structured proof using the method *cases* in the form

```
cases "term" rule: name
```

where *name* is the name of a valid case rule. The method prepares the rule by substituting the specified *term* for the first unknown in the assumptions, and applies the rule to the first goal in the goal state.

Additionally, the method creates a named context for every goal resulting from the rule application. The context contains (only) the variables and assumptions specified in the corresponding case in the case rule. For the most general form depicted above the context for the *i*th case contains the variables $x_{i1} \ \ldots \ x_{ipi}$ and the assumptions $Q_{i1} \ \ldots \ Q_{iqi}$. No term abbreviation *?case* is defined, because the conclusion of every new goal is the same as that of the original goal, thus the existing abbreviation *?thesis* can be used instead.

The names used for the contexts created by the *cases* method can be specified by attributing the case rule. Therefore, predefined case rules often create cases with names which are easy to understand by a proof writer.

In Isabelle HOL (see Section 5) the rule $[\![?Q \Longrightarrow ?P;\ \neg\ ?Q \Longrightarrow ?P]\!] \Longrightarrow ?P$ is available under the name *case_split*. It is attributed to use the case names *True* and *False*. Note that these are names, not the constants for the values of type *bool*.

Together, a structured proof for the goal $(x::nat) < c \Longrightarrow n*x \le n*c$ with case splitting may have the form

```
proof (cases "n = 0" rule: case_split)
case True
...
show ?thesis ⟨proof⟩
next
case False
...
show ?thesis ⟨proof⟩
qed
```

The *cases* method adds the assumptions $n=0$ and $n\neq 0$ respectively to the goals of the cases, the **case** statements add them as assumed facts to the local context, so that they are part of the rule exported by the **show** statement and match the assumption in the corresponding goal.

Note that the **case** statement adds only the assumptions originating from the case rule. The other assumptions in the original goal (here $x < c$) must be added to the context in the usual ways (see Section 2.2.8) if needed for the proof.

Often a case rule has only one unknown in the case assumptions. If there are more, several terms may be specified in the *cases* method for preparing the rule. If less terms are specified than there are unknowns in the case assumptions the resulting goals will contain unbound unknowns which must be instantiated in the rest of the proof (e.g., by method *drule*). If more terms are specified an error is signaled.

The *cases* method treats input facts like the empty method (see Section 2.3.1) by inserting them as assumptions into the original goal before splitting it. Therefore it can be used both in proof scripts, where facts are stored as rule assumptions in the goal state, and in structured proofs where facts are stored in the proof context and are input to the initial methods of subproofs.

However, if the *cases* method is specified in the form

```
cases "term"
```

without explicitly naming a case rule and the first input fact has the form of a case rule, it is used as case rule to split the goal and create the named cases. Therefore in a proof the example goal can be proved as local fact by inputting (see Section 2.2.6) the case rule in the form

```
from case_split
have "(x::nat) < c ⟹ n*x ≤ n*c"
proof (cases "n = 0")
...
```

Like the *rule* method (see Section 2.3.3) the *cases* method supports automatic rule selection for the case rule, if no case rule is specified or input to the method. Normally the rule is selected according to the type of the specified *term*. In Isabelle HOL (see Section 3) most types have an associated case rule. Only case rules with a single unknown in the case assumptions can be automatically selected in this way.

The rule *case_split* is associated with type *bool*. Therefore the example sub proof shown above also works without inputting the method to the **have** statement, because then it is selected automatically because the term *n = 0* has type *bool*.

The proof writer may not know the case names specified by an automatically selected case rule. However, they can be determined using the command **print_cases** or from the proof skeleton which is displayed in the interactive editor when the cursor is positioned on the *cases* method (see Section 2.4.1).

Alternative Syntax for Case Rules

A case rule as described above always uses an unknown *?P* (or with any other name) as conclusion and as conclusion in all assumptions. It is used technically to preserve the original conclusion when a goal is split by applying the rule. Therefore Isabelle supports an alternative syntax for specifying case rules as theorems which omits the variable for this unknown and specifies only the information which becomes the content of the named cases.

In a theorem in specified in structured form using **shows** (see Section 2.1.7) the part of the form **shows** *"C"* may alternatively be specified in the form

obtains C_1 | ... | C_k

where every case C_i has the form

x_{i1} ... x_{ipi} **where** "Q_{i1}" ... "Q_{iqi}"

As usual, the variables x_{i1} ... x_{ipi} and the propositions Q_{i1} ... Q_{iqi} may be grouped by *and*, for every variable group a type may be specified and the proposition groups may be named. The keywords and the | separators belong to the outer syntax, the propositions must be individually quoted.

This form is mainly equivalent to

shows "$\llbracket P_1;\ \ldots;\ P_k\rrbracket \Longrightarrow$ *thesis*"

where every P_i is a rule

$\bigwedge x_{i1} \ldots x_{ipi}.\ \llbracket Q_{i1};\ldots;Q_{iqi}\rrbracket \Longrightarrow$ *thesis*

which is exactly the general form of a case rule stated by the **shows** clause, using, after proof, the unknown *?thesis* for all conclusions.

For its own proof the **obtains** form creates the same goal as the **shows** form, but additionally it adds all P_i as assumed facts to the outermost proof context and names this set *that*.

When the theorem is applied as case rule by the *cases* method the named context created for case C_i is simply named *i*. An explicit name may be specified for it by using the extended form

($name_i$) x_{i1} ... x_{ipi} **where** "Q_{i1}" ... "Q_{iqi}"

For its own proof the **obtains** form uses $name_i$, if present, as additional name for P_i in its proof context.

If a case C_i has no bound variables it has simply the form

"Q_{i1}" ... "Q_{iqi}"

which omits the keyword **where**, also if an explicit name is specified.

As an example, the rule *case_split* may be defined and proved as

theorem *case_split:*
  obtains *(True) "Q" | (False) "¬ Q"*
  by *blast*

using the case names *True* and *False*, as described above, and using the *blast* method (see Section 2.3.7) for the proof.

There is also a statement for stating a case rule on the fly in a structured proof. It has the form

consider C_1 | ... | C_k ⟨*proof*⟩

where the cases C_i are as above. It is mainly equivalent to the statement

have "⟦P_1; ...; P_k⟧ ⟹ *thesis*" ⟨*proof*⟩

with P_i as above and is thus also a goal statement.

However, it is not possible to introduce additional unknowns in the P_i in a **consider** statement. All variables occurring free in the P_i are assumed to be bound in the context and are not converted to unknowns at the end of the statement. Therefore the statement cannot be used to define general case rules like *case_split* which contains the additional unknown *?Q*. It can only be used to state that specific propositions cover all (remaining) possibilities in the local proof context. They need not cover all globally possible cases, if some cases have already been excluded using locally assumed or proved facts, only the remaining possibilities need to be covered.

Since case rules can be input as fact to a proof by case based reasoning, fact chaining can be used to immediately apply a locally defined case rule in a subsequent subproof. This yields the pattern

```
consider C_1 | ... | C_k ⟨proof⟩
then have "prop"
proof cases
  case 1 ... show ?thesis ⟨proof⟩
next
...
  case k ... show ?thesis ⟨proof⟩
qed
```

using the default case names. The first ⟨*proof*⟩ proves that the cases cover all local possibilities, the other ⟨*proof*⟩s prove the stated proposition, each using one of the cases C_i as additional assumption. The *cases* method is always applied without arguments, since there are no additional unknowns in the C_i which can be instantiated.

If the example goal *(x::nat) < c ⟹ n*x ≤ n*c* should be stated for locally fixed variables, the cases *n=0* and *n≠0* can be proved to cover all possibilities and then used as cases by the statements

```
fix x c n::nat
consider (Zero) "n = 0" | (Notzero) "n ≠ 0" by blast
then have "(x::nat) < c ⟹ n*x ≤ n*c"
```

proof *cases*
  case *Zero* ... **show** *?thesis* ⟨*proof*⟩
next
  case *Notzero* ... **show** *?thesis* ⟨*proof*⟩
qed

2.4.4 Elimination

The proof technique of "(generalized) elimination" can be seen as a combination of applying a destruction rule (see Section 2.3.4) and an optional case split. It removes an assumption with a function application from a goal and splits the rest into cases with additional assumptions.

The Method *erule*

Like destruction rule application and case splitting such a step can be implemented by applying a rule in a specific way to a goal. This is done by the method

erule name

where *name* is the name of a valid rule. The method only affects the first goal. If that goal has the form $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ and the rule referred by *name* has the form $[\![RA_1;\ldots;RA_m]\!] \Longrightarrow RC$ the method first unifies the first assumption RA_1 in the rule with the first assumption A_k of the $A_1 \ldots A_n$ which can be unified with RA_1 *and* it unifies the rule conclusion RC with the goal conclusion C. That yields the specialized rule $[\![RA_1';\ldots;RA_m']\!] \Longrightarrow RC'$ where RA_1' is syntactically equal to A_k, RC' is syntactically equal to C and every RA_j' with $j > 1$ results from RA_j by substituting unknowns by the same terms as in RA_1' and RC'. If the goal does not contain unknowns (which is the normal case), A_k and C are not modified by the unifications. If no A_i unifies with RA_1 or C does not unify with RC the method cannot be executed on the goal state and an error is signaled. Otherwise the method replaces the goal by the $m-1$ new goals $[\![A_1;\ldots;A_n]\!] \Longrightarrow RA_j'$ for $j > 1$.

If the rule has the form $RA \Longrightarrow RC$ with only one assumption a successful application with *erule* removes the goal from the goal state. If the rule is a formula RC without any assumptions the method cannot be applied to the goal state and an error is signaled. If an assumption RA_j is a rule with own assumptions, these assumptions are appended to the assumptions in the resulting goal, as described for the *rule* method in Section 2.3.3.

As for rules applied by *frule* or *drule* (see Section 2.3.4) the first assumption RA_1 in the rule is called the "major premise" in this context.

The method *erule* does not support input facts.

Elimination Rules

An elimination rule is a generalized combination of a destruction rule and a case rule. It has the specific form

$\llbracket RA_1;\ \ldots;\ RA_n\rrbracket \Longrightarrow\ ?P$

where the first assumption contains the application of a specific function *f* (perhaps written using an operator name) which may only occur in different form in the other assumptions. The conclusion is a single unknown which must not occur in the first assumption and may only occur as conclusion in other assumption (like in an assumption in a case rule).

When an elimination rule is applied to a goal by the *erule* method, it removes ("eliminates") the function application from an assumption in the goal or it replaces it by a different form, so that it may be removed in several steps. Depending on the form of the other assumptions the resulting goals have either the same conclusion as the original goal or are unrelated to it. Therefore such an application of an elimination rule can be seen as a forward reasoning step, possibly with case splitting.

As an example consider the elimination rule specified as

theorem *elimexmp:* $"\llbracket(x::nat) \le c;\ x < c \Longrightarrow P;\ x = c \Longrightarrow P\rrbracket \Longrightarrow P"$

It replaces an assumption $x \le c$ by two cases with assumptions $x < c$ and $x = c$. If applied to the goal $(x::nat) \le 5 \Longrightarrow C$ by

apply *(erule elimexmp)*

it replaces the goal by the two goals

$x < 5 \Longrightarrow C$
$x = 5 \Longrightarrow C$

Analogous to the *intro* set for introduction rules there is an internal fact set *elim* for elimination rules. It is used by some automatic methods, however, it is not used for automatically selecting rules for *erule*.

If the cursor in the text area is positioned in a proof, elimination rules applicable to the first goal in the goal state of the enclosing proof can be searched using the keyword *elim* as criterion for a search by **find_theorems** or in the Query panel, as described in Section 2.1.7. It finds all named facts which can be applied by the *erule* method to the first goal, i.e., the major premise unifies with a goal assumption and the conclusions unify as well.

Elimination rule application can be iterated by the method

elim $name_1 \ldots name_n$

where $\mathtt{name}_1 \; \ldots \; \mathtt{name}_n$ refer to valid rules. It iterates applying rules by *erule* from the given set to a goal in the goal state as long as this is possible. It is intended to be used with elimination rules. Then it automatically eliminates functions from assumptions in the goals as much as possible with the given rules. Note that the method does not use the *elim* set.

Every destruction rule $[\![RA_1;\ldots;RA_n]\!] \Longrightarrow C$ can be re-stated as the elimination rule $[\![RA_1;\ldots;RA_n;C \Longrightarrow ?P]\!] \Longrightarrow ?P$. If that is applied by *erule* it has the same effect as if the original rule is applied by method *drule*.

Every case rule $[\![Q_1 \Longrightarrow ?P;\ldots;Q_k \Longrightarrow ?P]\!] \Longrightarrow ?P$ can be re-stated as the elimination rule $[\![Q;\; [\![Q;Q_1]\!] \Longrightarrow ?P;\ldots;[\![Q;Q_k]\!] \Longrightarrow ?P]\!] \Longrightarrow ?P$. If that is applied by *erule* to a goal with at least one assumption it has (except for assumption reordering) the same effect as if the original rule is applied by method *cases*.

Note, however, that *erule* does not create named contexts for the resulting cases.

Alternative Syntax for Elimination Rules

The alternative syntax available for case rules described in Section 2.4.3 can be extended for elimination rules. This requires that all assumptions which do not contain the unknown for the conclusion (this is at least the major premise) are grouped together at the beginning of all assumptions. Then an elimination rule specified as

theorem "$[\![RA_1;\ldots;RA_m;\; P_1;\; \ldots;\; P_k]\!] \Longrightarrow P$" ⟨*proof*⟩

where every P_i is a rule of the form

$\bigwedge x_{i1}\ldots x_{ipi}.\; [\![Q_{i1};\ldots;Q_{iqi}]\!] \Longrightarrow P$

and the variable P does not occur in the $RA_1 \; \ldots \; RA_m$ can be specified using the alternative syntax

theorem
assumes "RA_1" ... "RA_m"
obtains C_1 | ... | C_k
⟨*proof*⟩

where every C_i is

$x_{i1} \; \ldots \; x_{ipi}$ **where** "Q_{i1}" ... "Q_{iqi}"

The major premise and possibly other additional assumptions are specified using **assumes**, then the assumptions for the cases are specified using **obtains**. As usual, the set $RA_1,\; \ldots,\; RA_m$ is automatically named `assms` and the set of the P_i is automatically named *that* in the outermost proof context of the theorem, additional names may be specified explicitly.

The example rule *elimexmp* from the previous section can alternatively be specified as

```
theorem
assumes "(x::nat) ≤ c"
obtains "x < c" | "x = c"
⟨proof⟩
```

Elimination in Structured Proofs

The method *erule* is intended to be used in proof scripts, therefore it does not process input facts and does not create named cases. In structured proofs elimination can be done using the *cases* method.

The *cases* method applies a rule by elimination, if the rule is attributed by *[consumes 1]*. This means the rule will "consume" one assumption by unifying it with its major premise and removing it. A rule may be attributed upon definition in the form

```
theorem [consumes 1]: "prop" ⟨proof⟩
```

or it may be attributed on the fly when it is applied by the *cases* method:

```
cases "term" rule: name[consumes 1]
```

Since the *cases* method is intended to be used in structured proofs where facts are stored in the proof context it does not unify the rule's major premise with an assumption in the goal, it unifies it with the first input fact (possibly after the rule itself if not specified as method argument).

Now the rule *elimexmp* from the previous sections can be defined in the form

```
theorem elimexmp[consumes 1]:
  assumes "(x::nat) ≤ c"
  obtains (lower) "x < c" | (equal) "x = c"
  ⟨proof⟩
```

naming the cases by *lower* and *equal*. Then an example for an application in a structured proof with cases is

```
theorem "C" if "(x::nat) ≤ 5"
  using that
proof (cases rule: elimexmp)
  case lower ... show ?thesis ⟨proof⟩
next
  case equal ... show ?thesis ⟨proof⟩
qed
```

Note the use of the structured theorem form which puts the assumption `(x::nat) ≤ 5` into the proof context and names it `that` so that it can be input by **using** `that` into the structured proof and into its initial method `cases` which consumes it.

2.4.5 Induction

With induction a goal is proved by processing "all possible cases" for certain values which occur in it. If the goal can be proved for all these cases and the cases cover all possibilities, the goal holds generally. A specific technique is to assume the goal for some values and then prove it for other values. In this way it is possible to cover infinite value sets by proofs for only a finite number of values and steps from values to other values.

Perhaps the best known example of induction is a proposition which is proved for the natural number `0` and the step from a number `n` to its successor `n+1`, which together covers the whole infinite set of natural numbers.

As a (trivial) example consider the proposition `0≤n`. To prove that it is valid for all natural numbers `n` we prove the "base case" where `n` is `0`, which is true because `0≤0`. Then we prove the "induction step", by assuming that `0≤n` (the "induction hypothesis") and proving that `0≤n+1` follows, which is true because addition increases the value.

Induction Rules

Like for case based reasoning (see Section 2.4.3) a goal is split into these cases by applying a meta-rule. For induction the splitting can be done by a meta-rule of the form

$[\![P_1\ ;\ \ldots;\ P_n\ ]\!] \Longrightarrow\ ?P\ ?a$

where every P_i is a rule of the form

$\bigwedge y_{i1}\ \ldots\ y_{ip_i}.\ [\![Q_{i1};\ \ldots;\ Q_{iq_i}]\!] \Longrightarrow\ ?P\ term_i$

where the assumptions Q_{ij} may contain the unknown `?P` but no other unknowns, in particular not `?a`. A rule for a base case usually has no bound variables y_{ij} and no assumptions Q_{ij}, at least the Q_{ij} do not contain `?P`. The remaining rules mostly have only a single assumption Q_{ij} which contains `?P`.

Note that the unknown `?a` only occurs once in the conclusion of the meta-rule and nowhere else. Like the case rules induction rules must be "prepared" for use, this is done by replacing `?a` by a specific term `term`. This is the term for which all possible cases shall be processed in the goal. It must have the same type as all $term_i$ in the P_i.

Usually, "all possible cases" means all values of the type of `term`, then `term` consists of a single variable which may adopt any values of its type. There are also other forms of induction where more complex terms are used, but they are not presented in this introduction, refer to other Isabelle documentations for them. In the following the unknown `?a` will always be replaced by a variable `x`.

When a prepared induction rule is applied to a goal `C` without bound variables and assumptions as described in Section 2.3.3, it unifies `?P x` with the conclusion `C`. This has the effect of abstracting `C` to a (boolean) function `PC` by identifying all places where `x` occurs in `C` and replacing it by the function argument. The function `PC` is then bound to the unknown `?P`, so that applying `?P` to the argument `x` again yields `C`. The function `PC` is the property to be proved for all possible argument values. Therefore the cases of the proof can be described by applying `?P` to the terms $term_i$ for the specific values in the rules P_i for the cases.

The application of the prepared rule results in the `n` goals

$\bigwedge y_{11} \ldots y_{1p1}.\ [\![Q_{11};\ \ldots;\ Q_{1q1}]\!] \Longrightarrow PC\ term_1$
$\ldots$
$\bigwedge y_{n1} \ldots y_{npn}.\ [\![Q_{n1};\ \ldots;\ Q_{nqn}]\!] \Longrightarrow PC\ term_n$

The induction rule is only valid if the terms $term_i$ cover all possible values of their associated type. If this has been proved the induction rule is available as a fact which can be applied. After preparing the induction rule for application, its conclusion `?P x` matches all propositions which contain the variable `x` in one or more copies. It depends on the P_i in the rule whether splitting a specific goal with the induction rule is useful for proving the goal.

The real power of induction rules emerges, when a Q_{ij} contains the unknown `?P`. Due to the type associated with `?P` it must be applied to an argument $term_{ij}$ of the same type as `x` and the $term_i$. Then the goal resulting from P_i states the property that if Q_{ij} holds when specialized to $PC\ term_{ij}$, `PC` holds for $term_i$ (an "induction step"). Thus, for covering the possible values of `x`, the step from $term_{ij}$ to $term_i$ can be repeated arbitrarily often which allows to cover some types with infinite value sets.

An induction rule for the natural numbers is

$[\![?P\ 0;\ \bigwedge y.\ ?P\ y \Longrightarrow\ ?P\ (y+1)]\!] \Longrightarrow\ ?P\ ?a$

P_1 is the base case, it has no variables and assumptions and only consists of the conclusion `?P 0`. P_2 binds the variable `y`, has one assumption `?P y` and the conclusion `?P (y+1)`. P_1 covers the value `0`, P_2 covers the step from a value `y` to its successor `y+1`, together they cover all possible values of type `nat`.

To apply the rule to the goal $0 \leq n$, it must be prepared by substituting the variable `n` for the unknown `?a`. Then the rule conclusion `?P n` is unified with the goal which abstracts the goal to the boolean function `PC =`

$(\lambda$`i. 0≤i)` and substitutes it for all occurrences of `?P`. This results in the rule instance $\llbracket$`(λi. 0≤i) 0; ⋀y. (λi. 0≤i) y ⟹ (λi. 0≤i) (y+1)`$\rrbracket \Longrightarrow$ `(λi. 0≤i) n`. By substituting the arguments in the function applications its assumption part yields the two goals

```
0≤0
⋀y. 0≤y ⟹ 0≤(y+1)
```

which correspond to the base case and induction step as described above.

Induction rules may even be more general than shown above. Instead of applying `?P` to a single argument it may have several arguments and the conclusion becomes `?P` $?a_1 \ldots ?a_r$. Also in the P_i every occurrence of `?P` then has r terms as arguments. Such an induction rule is valid if it covers all possible cases for all combinations of the r argument values. Finally, in addition to the assumptions P_i an induction rule may have assumptions about the argument `?a` or the arguments $?a_1 \ldots ?a_r$.

Note, however, that there is no alternative syntax for induction rules, such as the **obtains** form for case rules.

The *induction* Method

Induction can be performed in a structured proof using the method *induction* in the form

```
induction x rule: name
```

where *name* is the name of a valid induction rule. The method prepares the rule by substituting the specified variable *x* for the unknown `?a` and applies the rule to the first goal in the goal state.

Additionally, the method creates a named context for every goal resulting from the rule application. The context contains the variables and assumptions specified in the corresponding case in the induction rule. For the general form depicted above the context for the *i*th case contains the variables $y_{i1} \ldots y_{ipi}$ and the assumptions $Q_{i1}; \ldots; Q_{iqi}$. The term abbreviation `?case` is defined for the case conclusion *PC* $term_i$ which is to be proved for the case.

The *induction* method treats input facts like the empty method (see Section 2.3.1) and the *cases* method (see Section 2.4.3) by inserting them as assumptions into the original goal before splitting it.

Also like the *cases* method the *induction* method supports automatic rule selection for the induction rule. This is only possible if `?P` is applied to a single argument, which means that only one variable is specified in the method:

```
induction x
```

Then the rule is selected according to the type of `x`. In Isabelle HOL (see Section 3) most types have an associated induction rule.

The rule `⟦?P True; ?P False⟧ ⟹ ?P ?a` is associated with type `bool`. Therefore induction can be applied to every proposition which contains a variable of type `bool`, such as the goal `b ∧ False = False`. Applying the method

```
induction b
```

will split the goal into the goals

```
False ∧ False = False
True ∧ False = False
```

which cover all possible cases for `b`. Here, the type has only two values, therefore induction is not really needed.

Like for the `cases` method (see Section 2.4.3) the names used for the contexts created by the `induction` method can be specified by attributing the induction rule. They can be determined from the proof skeleton which is displayed in the interactive editor when the cursor is positioned on the `induction` method (see Section 2.4.1).

If the induction rule `⟦?P 0; ⋀y. ?P y ⟹ ?P (y+1)⟧ ⟹ ?P ?a` for the natural numbers has been proved and named `induct_nat` with case names `Start` and `Step`, a structured proof for the goal `0≤n` may have the form

```
proof (induction n rule: induct_nat)
case Start
...
show ?case ⟨proof⟩
next
case Step
...
show ?case ⟨proof⟩
qed
```

The `induction` method creates the named contexts `Start` and `Step`. The former has no local variables and assumptions and binds `?case` to the proposition `0≤0`, the latter has the local variable `y`, the assumption `0≤y` named `Step` and binds `?case` to the proposition `0 ≤ y + 1`.

If the rule `induct_nat` has been associated with type `nat` the rule specification may be omitted in the method:

```
proof (induction n)
...
```

Case Assumption Naming and the *induct* Method

As usual, the **case** statement uses the case name as name for the assumptions Q_{i1} ... Q_{iqi} in the *i*th case or an explicit name may be specified for them (see Section 2.4.1). Additionally, the *induction* method arranges the named context for a case so that the set of assumptions is split into those which in the rule contain the unknown *?P* and those which do not. These sets are separately named, so that they can be referenced individually.

The set of assumptions which originally contained *?P* now contain an application of *PC* to a value $term_{ij}$ and allow the step from this value to value $term_i$ by induction. These assumptions are called “induction hypotheses” and are named *"cname.IH"* where *cname* is the case name or the explicit name for the case assumptions. The other assumptions are independent from *PC*, they are additional hypotheses and are named *"cname.hyps"*. Both forms of names must be enclosed in quotes because the dot is not a normal name constituent.

For an example consider the induction rule $[\![?P\ 0;\ ?P\ 1;\ \bigwedge y.\ [\![y\geq 1;\ ?P\ y]\!] \Longrightarrow ?P\ (y+1)]\!] \Longrightarrow ?P\ ?a$ with an additional base case for the value *1* and a step which always starts at value *1* or greater. If applied to the goal $0\leq n$ the *induction* method produces the three goals

$0\leq 0$
$0\leq 1$
$\bigwedge y.\ [\![y\geq 1;\ 0\leq y]\!] \Longrightarrow 0\leq (y+1)$

If the default case name *3* is used for the third case, the induction hypothesis $0\leq y$ is named *"3.IH"* and the additional hypothesis $y\geq 1$ is named *"3.hyps"*.

There is a method *induct* which behaves like *induction* with the only difference that it does not introduce the name *"cname.IH"*, it uses *"cname.hyps"* for all assumptions Q_{i1} ... Q_{iqi}, whether they contain *?P* or not.

Goals with Assumptions

If the *induction* method would apply the prepared induction rule in the same way as the *rule* method to a goal $[\![A_1;\ \ldots;\ A_m]\!] \Longrightarrow C$ with assumptions it would unify *?P x* only with the conclusion *C* and copy the assumptions A_1, ..., A_m to all resulting goals unchanged. However, if *x* also occurs in one or more of the A_l this connection with *C* is lost after applying the prepared induction rule.

Consider the goal

$4 < n \Longrightarrow 5 \leq n$

which is of the form

$A \Longrightarrow C$

When applying the prepared induction rule for the natural numbers $[\![?P\ 0;\ \bigwedge y.\ ?P\ y \Longrightarrow\ ?P\ (y+1)]\!] \Longrightarrow\ ?P\ n$ in the way of the *rule* method the conclusion will be matched which leads to the abstracted function $PC \equiv (\lambda i.\ 5 \le i)$ and the resulting goals are

$4 < n \Longrightarrow 5 \le 0$
$\bigwedge y.\ [\![4 < n;\ 5 \le y]\!] \Longrightarrow 5 \le (y+1)$

where the first goal is invalid. Although the second goal is valid, it shows that the relation between the variable n in the assumption and the variable y used in the induction rule has been lost.

For a goal with assumptions every occurrence of $?P$ in the rule, applied to a specific term must be replaced by PC applied to the term together with all assumptions which must also be adapted to the same term. Therefore the *induction* and *induct* methods work in a different way. They unify $?P\ x$ with the conclusion C and separately with every assumption A_l and thus additionally determine an abstracted function PA_l for every A_l. Then they replace every $?P\ term$ in a P_i or in a Q_{ij} in the rule by an application $PC\ term$ together with assumptions $PA_1\ term;\ \ldots;\ PA_m\ term$.

The assumptions for a direct occurrence of $?P\ term_i$ as conclusion in a P_i are added after all Q_{ij}, so that the ith goal created by the *induction* method now has the form

$\bigwedge\ y_{i1}\ \ldots\ y_{ipi}.\ [\![Q_{i1};\ \ldots;\ Q_{iqi};\ PA_1\ term_i;\ \ldots;\ PA_m\ term_i]\!] \Longrightarrow PC\ term_i$

Additionally, the assumptions for occurrences of a $?P\ term$ in a Q_{ij} must be added which usually can be done by replacing Q_{ij} by the rule Q_{ij}' of the form $[\![PA_1\ term;\ \ldots;\ PA_m\ term]\!] \Longrightarrow Q_{ij}''$ where Q_{ij}'' results by substituting $?P\ term$ in Q_{ij} by $PC\ term$. If $?P$ is applied to several different terms in Q_{ij}, several sets of corresponding assumptions are added in Q_{ij}'.

Moreover, the *induction* method (and also the *induct* method) arranges the named contexts in a way that the assumptions $PA_1\ term_i;\ \ldots;\ PA_m\ term_i$ which originate from the goal are named by "cname.prems" and can thus be referenced separate from the Q_{ij}' which are named "cname.hyps" and possibly "cname.IH" as described above.

In the example above the *induction* method additionally unifies $?P\ n$ with the assumption $4 < n$ which yields the abstracted function $PA \equiv (\lambda i.\ 4<i)$ and produces the goals

$4 < 0 \Longrightarrow 5 \le 0$
$\bigwedge y.\ [\![4 < y \Longrightarrow 5 \le y;\ 4 < (y+1)]\!] \Longrightarrow 5 \le (y+1)$

Here $4 < (y+1)$ results from applying PA to $(y+1)$ and $4 < y$ results from adding PA applied to y as assumption to the assumption $?P\ y$ from the rule.

If the default case name *2* is used for the second case, the case assumption $4 < y \Longrightarrow 5 \le y$ will be named *"2.IH"* and the case assumption *4 < (y+1)* will be named *"2.prems"* by the **case** statement.

Like the *cases* method the methods *induction* and *induct* insert input facts as assumptions into the goal before they process the goal assumptions in the described way. Therefore both methods can be used both in proof scripts, where facts are stored as rule assumptions in the goal state, and in structured proofs where facts are stored in the proof context and are input to the initial methods of subproofs.

Other than for the *cases* method it is not possible to pass the induction rule as input fact to the methods *induction* and *induct*.

Induction with Elimination

Like case rules, induction rules can be extended to include elimination (see Section 2.4.4). Such induction rules have an additional first assumption which is used as major premise to unify with a goal assumption and eliminate it, before processing the remaining goal assumptions and splitting the goal into cases.

Like case rules such extended induction rules must be attributed by *[consumes 1]*, then the methods *induction* and *induct* apply elimination before doing the rest of processing.

As example consider the rule $[\![?a \ge 42;\ ?P\ 42;\ \bigwedge y.\ ?P\ y \Longrightarrow ?P\ (y+1)]\!] \Longrightarrow ?P\ ?a$. It uses *42* as its base case and can thus only prove a property for numbers equal to or greater than *42*. The major premise restricts *?a* to these cases. If this rule has been proved and named *induct_nat42* it may be applied with elimination as initial method of a structured proof for the goal $4 < n \Longrightarrow 5 \le n$ by

proof *(induction n rule: induct_nat42[consumes 1])*

If the fact $n \ge 42$ is available as first input fact the application will be successful, consume that fact, and split the goal into the goals

$4 < 42 \Longrightarrow 5 \le 42$
$\bigwedge y.\ [\![4 < y \Longrightarrow 5 \le y;\ 4 < (y+1)]\!] \Longrightarrow 5 \le (y+1)$

In contrast to the *cases* method the methods *induction* and *induct* will also consume the first assumption present in the goal, if it unifies with the major premise of the induction rule and no input facts are present.

Arbitrary Variables

If the goal contains bound variables, i.e., is of the form $\bigwedge x_1 \ldots x_k.\ [\![A_1;\ \ldots;\ A_m]\!] \Longrightarrow C$ these variables may occur in the assumptions $A_1, \ldots, A_m$

and the conclusion *C*. When these are instantiated for the occurrences of *?P* in the rule as described above, every such instance needs its own separate copy of bound variables $x_1 \dots x_k$.

Consider the goal

```
⋀c. n<c ⟹ 2*n < 2*c
```

When applying the prepared induction rule for the natural numbers ⟦*?P 0; ⋀y. ?P y ⟹ ?P (y+1)*⟧ ⟹ *?P n* in the way described above, the variable *c* must be bound separately for the occurrences *?P y* and *?P (y+1)* because it need not denote the same value in both positions. This can be accomplished by creating the goals

```
⋀c. 0<c ⟹ 2*0 < 2*c
⋀y c. ⟦⋀c. y<c ⟹ 2*y < 2*c; y+1 < c⟧ ⟹ 2*(y + 1) < 2*c
```

with two nested bindings of *c*.

The methods *induction* and *induct* treat explicit bindings of variables in the goal in this way. However, if variables are not bound explicitly in the goal (i.e., they are free in the goal), there are two possible meanings: in a theorem all variables are implicitly bound and therefore need a separate binding for every occurrence of *?P*, whereas in a local goal statement variables may have been fixed in the enclosing context, then they must denote the same value for every occurrence of *?P* and must not be bound separately. Since it is difficult for Isabelle to determine the correct treatment of free variables automatically, it may be specified explicitly by the proof writer. The free variables which do not denote a fixed value from the context but an “arbitrary” value used only locally in the goal can be specified to the *induction* method in the form

induction x arbitrary: $x_1 \dots x_k$ *rule: name*

and in the same form for the *induct* method.

In particular, if a theorem or goal statement is specified in structured form the free variables are not bound in the goal but in the outermost proof context (see Section 2.2.1) and the goal only consists of the conclusion *C*. To apply induction as initial proof method the assumptions must be input to it and the variables must be specified as arbitrary:

theorem
  fixes $x_1 \dots x_k$
  assumes "A_1" ... "A_m"
  shows "*C*"
using *assms*
proof *(induction ... arbitrary:* $x_1 \dots x_k$ *...)* ... **qed**

Chapter 3

Isabelle HOL Basics

The basic mechanisms described in Chapter 2 can be used for working with different "object logics". An object logic defines the types of objects available, constants of these types, and facts about them. An object logic may also extend the syntax, both inner and outer syntax.

The standard object logic for Isabelle is the "Higher Order Logic" HOL, it covers a large part of standards mathematics and is flexibly extensible. This chapter introduces some basic mechanisms which are available in HOL for arbitrary types.

The abbreviation "HOL" is used in the logics community to denote the general concept of higher order logics. In this document we use it specifically to denote the implementation as object logic in Isabelle which is also named HOL.

3.1 Predicates and Relations

Basically HOL uses the type `'a ⇒ 'b` of functions provided by Isabelle (see Section 2.1.3) to represent predicates and relations in the usual way.

3.1.1 Predicates

A predicate on values of arbitrary types t_1, ..., t_n is a function of type $t_1 \Rightarrow \ldots \Rightarrow t_n \Rightarrow$ `bool`. It may be denoted by a lambda term $\lambda x_1 \ldots x_n$. `bterm` where `bterm` is a term of type `bool` which may contain free occurrences of the variables x_1, ..., x_n. Note that also a predicate defined as a named constant `P` can always be specified by the lambda term $\lambda x_1 \ldots x_n$. `P` $x_1 \ldots x_n$.

Note that the concept of predicates actually depends on the specific HOL type `bool`, which is introduced in Section 5.1.

3.1.2 Unary Predicates and Sets

Unary predicates of type `t ⇒ bool` are equivalent to sets of type `t set`. There is a 1-1 correspondence between a predicate and the set of values for which the predicate value is `True`. Actually, the HOL type constructor `set` described in Section 5.4 is defined based on this correspondence. See that section for more information about denoting and using sets.

As a convention HOL often provides a predicate and rules about it in both forms as a set named `name` and as a predicate named `namep` or `nameP`. Then usually every fact `F` containing such predicates in set form can be converted to the corresponding fact containing them in predicate form by applying the attribute `to_pred` as in `F[to_pred]` and vice versa by applying the attribute `to_set`.

3.1.3 Relations

A binary relation between values of arbitrary types t_1 and t_2 is a binary predicate of type t_1 ⇒ t_2 ⇒ `bool`. As binary functions the relations in HOL often have an alternative operator name of the form `(op)` which supports specifying applications in infix form `x op y` (see Section 2.1.3)..

By partial application (see Section 2.1.3) the first argument of a relation `R` can be fixed to yield the unary predicate `(R x)` on the second argument. For operators this must be done using the operator name in the form `((op) x)`, partial application cannot be written by omitting an argument on one side of the infix operator.

Since the unary predicate `(R x)` is equivalent to a set, every binary relation of type t_1 ⇒ t_2 ⇒ `bool` is equivalent to a set-valued function of type t_1 ⇒ (t_2 `set`). It maps every value of type t_1 to the set of related values of type t_2. HOL extends the convention described above and often provides relations named `namep` or `nameP` also as set-valued functions named `name`.

Note that HOL introduces the specific type constructor `rel` (see Section 5.8), where the values are equivalent to binary relations.

More generally, `n`-ary relations for `n > 2` are directly represented by `n`-ary predicates. Every `n`-ary relation is equivalent to an `(n-1)`-ary set-valued function.

3.2 Equality, Orderings, and Lattices

3.2.1 The Equality Relation

HOL introduces the equality relation as a function

```
HOL.eq :: 'a ⇒ 'a ⇒ bool
```

with the alternative operator name *(=)* for infix notation.

Inequality can be denoted by

```
not_equal :: 'a ⇒ 'a ⇒ bool
```

with the alternative operator name *(≠)* for infix notation.

Both functions are polymorphic and can be applied to terms of arbitrary type, however, they can only be used to compare two terms which have the same type. Therefore the proposition

```
True ≠ 1
```

is syntactically wrong and Isabelle will signal an error for it.

Moreover, in a term such as *term*$_1$ *=* *term*$_2$ or *term*$_1$ ≠ *term*$_2$ no type information can be derived for *term*$_1$ and *term*$_2$ other than that they must have the same type. There may be relations with the same semantics but for operands of a specific type, then it is possible to derive the operand types. An example is the relation *iff* with operator name *(⟷)* which is equal to *(=)* but only defined for operands of type *bool*. Therefore for the term *term*$_1$ ⟷ *term*$_2$ HOL automatically derives that *term*$_1$ and *term*$_2$ have type *bool*.

3.2.2 The Ordering Relations

HOL also introduces two ordering relations as functions

```
less :: 'a ⇒ 'a ⇒ bool
less_eq :: 'a ⇒ 'a ⇒ bool
```

with the alternative operator names *(<)* and *(≤)* for infix notation and the abbreviations *greater* and *greater_eq* for reversed arguments with operator names *(>)* and *(≥)*.

Based on these relations HOL also introduces the functions

```
min :: 'a ⇒ 'a ⇒ 'a
max :: 'a ⇒ 'a ⇒ 'a
```

in the usual way, i.e. if *a≤b* then *min a b* is *a*, otherwise *b*. HOL also introduces the functions

```
Min :: 'a set ⇒ 'a
Max :: 'a set ⇒ 'a
```

for the minimum or maximum of all values in a set.

All these functions are polymorphic and can be applied to terms of arbitrary type. Even if the values of a type are not ordered, an application of an ordering relation or minimum/maximum function to them is a correct term.

However, in that case the resulting value is underspecified, no information is available about it. Also, the value of `Min` and `Max` is underspecified if applied to an empty or infinite set.

Moreover, these are only syntactic definitions, no rules about orderings are implied by them. For some of its predefined types, such as type `nat`, HOL provides more specific specifications by overloading.

Like for `(=)` the type of $term_1$ and $term_2$ cannot be derived in a term such as $term_1$ `<` $term_2$. There are relations with the same semantics but specific operand types such as `(⊆)` for the relation `(≤)` on sets (see Section 5.4).

3.2.3 Lattice Operations

HOL introduces the two lattice operations

```
inf :: 'a ⇒ 'a ⇒ 'a
sup :: 'a ⇒ 'a ⇒ 'a
```

with the alternative operator names `(⊓)` and `(⊔)` for infix notation together with the lattice constants

```
top :: 'a
bot :: 'a
```

with the alternative names `(⊤)` and `(⊥)` (in the interactive editor available in the Symbols panel in the Logic tab).

Additionally there are the lattice operations for all values in a set (see Section 5.4)

```
Inf :: 'a set ⇒ 'a
Sup :: 'a set ⇒ 'a
```

with the alternative operator names `(⨅)` and `(⨆)` for prefix notation.

HOL does not provide the six alternative names automatically. To make them available, the command

unbundle `lattice_syntax`

must be used on theory level. It is available after importing the theory `Main` (see Section 1.1.2).

The lattice operations and constants are polymorphic but are not available for arbitrary types. They are overloaded only for those types which have a corresponding structure. For example, type `nat` has the `bot` value (which is equal to `0`), but no `top` value. If a lattice operation or constant is applied to an operator for which it is not available an error message of the form “No type arity ...” is signaled.

Like for equality and ordering relations, because the lattice operations and constants are overloaded it is not possible to derive the type for valid operands. Again, there are operands and constants with more specific operand types, such as `(∩)` for `(⊓)` on sets where HOL automatically derives the operand types.

3.3 Description Operators

A description operator selects a value from all values which satisfy a given unary predicate.

Description operators use "binder syntax" of the form `OP x. term`. Like a lambda term (see Section 2.1.3) it locally binds a variable `x` which may occur in `term`.

3.3.1 The Choice Operator

An arbitrary value satisfying the given predicate `λx. bterm` can be denoted by

```
SOME x. bterm
```

Only a single variable may be specified after `SOME`, however, like in a lambda term a type may be specified for it.

The value denoted by the term is underspecified in the sense of Section 2.1.3. The only information which can be derived for it is that it satisfies the predicate `λx. bterm`. If there is no value which satisfies the predicate not even this property may be derived.

The operator `SOME` is equivalent to the famous Hilbert choice operator. HOL includes the axiom of choice and provides the operator on this basis.

3.3.2 The Definite Description Operator

If only one value satisfies the given predicate `λx. bterm` this value can be denoted by

```
THE x. bterm
```

Like for `SOME` only a single variable may be specified with an optional type specification.

The value denoted by the term is also underspecified. However, after proving that there exists a value `v` which satisfies the predicate `λx. bterm` and that all values satisfying the predicate are equal it is possible to prove that `THE x.bterm = v`.

3.3.3 The Least and Greatest Value Operators

If the values satisfying a predicate `λx. bterm` are ordered, the least or greatest of these values can be denoted by the terms

```
LEAST x. bterm
GREATEST x. bterm
```

Only a single variable with an optional type specification is useful to be specified.

If the values satisfying the predicate are not ordered the value denoted by such a term is underspecified. The operators use the ordering relations `(≤)` and `(≥)` (see Section 3.2.2) which are applicable to values of arbitrary type. The operators are defined using `THE` to return the value `x` which satisfies the predicate and `x ≤ y` or `x ≥ y` holds, respectively, for all values `y` which also satisfy the predicate.

Also, if the values are ordered but there is no single least or greatest value among them the resulting value is underspecified. For example, the term `GREATEST n::nat. n > 0` is correct and denotes a value of type `nat`, although no information is available about it.

3.4 Undefined Value

HOL introduces the undefined value

```
undefined :: 'a
```

which is overloaded for arbitrary types. It is completely underspecified as described in Section 2.1.3, i.e., no further information is given about it.

Despite its name it is a well defined value for every type `'a`. It is typically used for values which are irrelevant, such as in the definition

definition `f :: "nat ⇒ nat"` **where** `"f x ≡ undefined"`

Although the function `f` looks like a completely undefined function, it is not possible to define true partial functions this way. Functions in Isabelle are always total. Function `f` maps every natural number to the (same) value `undefined`, which is of type `nat`, but it cannot be proved to be equal to a specific natural number such as `1` or `5`. However, since it is a single value the equality `f x = f y` holds for arbitrary `x` and `y` of type `nat`.

3.5 Let Terms

HOL extends the inner syntax for terms described in Section 2.1.3 by terms of the following form

$let\ x_1\ =\ term_1;\ \ldots;\ x_n\ =\ term_n\ in\ term$

where $x_1,\ \ldots,\ x_n$ are variables. The variable bindings are sequential, i.e., if $i<j$ variable x_i may occur in $term_j$ and denotes $term_i$ there. In other words, the scope of x_i are the terms $term_j$ with $i<j$ and the $term$. If x_i and x_j are the same variable, the binding of its second occurrence shadows the binding of the first and ends the scope of the first occurrence.

Let terms are useful to introduce local variables as abbreviations for sub-terms.

The let term specified above is an alternative syntax for the nested let terms

$let\ x_1\ =\ term_1\ in\ (\ldots\ (let\ x_n\ =\ term_n\ in\ term)\ \ldots)$

and a single let term

`let x = term' in term`

is an alternative syntax for the function application term

`Let term' (λx. term)`

Here `Let` is the polymorphic function

`Let :: 'a ⇒ ('a ⇒ 'b) ⇒ 'b ≡ λx f. f x`

which simply applies its second argument to its first argument.

Occurrences of let terms are usually not automatically resolved by substituting the bound term for the variable. Therefore a proposition like `(let x = a+b in (x*x)) = ((a+b)*(a+b))` cannot be proved by the simplifier (or other methods including simplification like `auto`, see Sections 2.3.6 and 2.3.7). To resolve it, the simplifier must be configured by adding the definitional equation of `Let` as `simp add: Let_def`.

3.6 Tuples

HOL supports type expressions of the form $t_1\ \times\ \ldots\ \times\ t_n$ for arbitrary types $t_1,\ \ldots,\ t_n$. They denote the type of n-tuples where the ith component has type t_i. Here the $\times$ is the “times operator” available in the interactive editor in the Symbols panel in the Operator tab. An alternative syntax is $t_1\ *\ \ldots\ *\ t_n$ using the ASCII star character. As an example the type `nat × bool` is the type of pairs of natural numbers and boolean values. The type `nat × 'a × bool` is the polymorphic type of triples of a natural number, a value of the arbitrary type denoted by the type parameter `'a`, and a boolean value. As usual, all these type expressions are part of the inner syntax.

Values for n-tuples of type $t_1\ \times\ \ldots\ \times\ t_n$ are denoted in inner syntax as terms of the form $(term_1,\ \ldots,\ term_n)$ where $term_i$ is a term of type t_i and the parentheses and the comma belong to the syntax.

3.6.1 Function Argument Tuples

Every `n`-ary function of type `t`$_1$ `⇒ ... ⇒ t`$_n$ `⇒ t` (see Section 2.1.3) is equivalent to a function of type `(t`$_1$ `× ... × t`$_n$`) ⇒ t` with `n`-tuples as argument values, which is the common way in mathematics to represent a function with `n` arguments. There is a 1-1 correspondence between functions of these two forms. The first form is called the "curried" form and the second form with tuples as arguments is called the "uncurried" form (named after Haskell Curry who introduced the first form for `n`-ary functions). Note that for unary functions both forms are the same.

Section 5.7 describes means to convert between both forms and other ways how to work with them.

3.6.2 Relations as Tuple Sets

For an `n`-ary relation of type `t`$_1$ `⇒ ... ⇒ t`$_n$ `⇒ bool` (see Section3.1.3) the uncurried form is a predicate of type `(t`$_1$ `× ... × t`$_n$`) ⇒ bool`. Since all arguments together are represented by a single tuple, this predicate is equivalent to a set of type `(t`$_1$ `× ... × t`$_n$`) set` (see Section3.1.2) where the elements in the set are tuples. This is the usual form of representing relations in mathematics.

HOL extends the convention described in Section 3.1.2 of providing relations named `namep` or `nameP` also as set-valued functions named `name` by alternatively using a tuple set named `name`.

3.7 Inductive Definitions

HOL supports inductive definitions of predicates as content in theories. An inductive definition defines a predicate by derivation rules which allow to derive the predicate values for some arguments from the values for other arguments. An inductive definition for a k-ary predicate has the form

inductive `name :: "t`$_1$ `⇒ ... ⇒ t`$_k$ `⇒ bool"`
where `P`$_1$ `| ... | P`$_n$

with derivation rules `P`$_i$. The type specification for the predicate may be omitted if it can be derived from the use of the defined predicate in the rules `P`$_i$.

3.7.1 The Defining Rules

The `P`$_i$ are derivation rules which may be specified in inner syntax of the form

$\bigwedge$ `x`$_{i1}$ `...` `x`$_{ipi}$`.` $[\![$`Q`$_{i1}$`;` `...;` `Q`$_{iqi}$$]\!]$ $\Longrightarrow$ `name term`$_{i1}$ `...` `term`$_{ik}$

where the conclusion is always an application of the defined predicate to `k` argument terms. The defined predicate `name` may occur in arbitrary ways in the rule assumptions `Q`$_{ij}$, but not in the argument terms `term`$_{ij}$. The rule separators `|` belong to the outer syntax, thus every rule must be separately quoted.

An example is the following inductive definition for an evenness predicate:

inductive *evn :: "nat ⇒ bool"* **where**
  "evn(0)" | "⋀n. evn(n) ⟹ evn(n+2)"

Alternative Rule Forms

As for other derivation rules on theory level (see Section 2.1.6) the explicit bindings of the variables `x`$_{i1}$`, ...,` `x`$_{ipi}$ are optional, variables occurring free in the assumptions or the conclusion are always automatically bound. As usual, types may be specified for (some of) the variables, so explicit bindings can be used as a central place for specifying types for the variables, if necessary. The equivalent example with omitted binding is

inductive *evn :: "nat ⇒ bool"* **where**
  "evn(0)" | "evn(n) ⟹ evn(n+2)"

Alternatively a rule `P`$_i$ may be specified in the structured form described in Section 2.1.6:

"name term$_{i1}$ *... term*$_{ik}$*"* **if** *"Q*$_{i1}$*" ... "Q*$_{iqi}$*"* **for** *x*$_{i1}$ *... x*$_{ipi}$

where the assumptions and variables may be grouped by **and** and types may be specified for (some of) the variable groups, however, no names may be specified for assumption groups, because no proof is specified for the rules. In this form the example is written

inductive *evn :: "nat ⇒ bool"* **where**
  "evn(0)" | "evn(n+2)" **if** *"evn(n)"*

The rules `P`$_i$ are introduction rules (see Section 2.3.3) for the defined predicate `name` in the sense that they introduce a specific application of `name`, possibly depending on other applications of `name`. An inductive definition turns the rules `P`$_i$ to valid facts "by definition" under the fact set name `name.intros`. Explicit names for (some of) the rules may be specified in the form

inductive *name :: "t*$_1$ *⇒ ... ⇒ t*$_k$ *⇒ bool"*
where *rname*$_1$*: P*$_1$ *| ... | rname*$_n$*: P*$_n$

Using this form the example can be written as

inductive *evn :: "nat ⇒ bool"* **where**
  zero: "evn(0)" | step: "evn(n) ⟹ evn(n+2)"

Note that the syntax of the defining rules only allows to specify when the defined predicate is *True*, not when it is *False*. In particular, a rule conclusion may not have the form of a negated application ¬ *name* $term_{i1}$ … $term_{ik}$.

Semantics of an Inductive Definition

To determine whether an application of the predicate to argument values yields the value *True* the defining rules P_i can be applied as introduction rules in backward reasoning steps (see Section 2.3.3) until the predicate is not present anymore. However, depending on the rules, this may not succeed. If the defining rules P_i would be the only information available about the defined predicate, it would be underspecified for such cases. The specific property of an *inductive* definition is the additional regulation that in all such cases the predicate value is *False*. In other words, if it is not possible to prove that the predicate value is *True* by using the defining rules it is defined to be *False*. This can happen in two ways.

In the first case there is no rule for which the conclusion unifies with the predicate application term. Consider the trivial inductive definition

inductive $uspec_1$ *:: "nat ⇒ bool"* **where** *"*$uspec_1$ *3" | "*$uspec_1$ *4"*

The rules only unify with applications of $uspec_1$ to the argument values *3* and *4*, for them it can be derived that the predicate value is *True*. For all other arguments the value is *False*.

In the example the value of *(evn 3)* is *False* because an application of the defining rule *step* as backwards reasoning step leads to *(evn 1)* which does not unify with the conclusion of either rule.

In the second case there are rule conclusions which unify with the predicate application term, but there is no finite sequence of backward rule application steps which removes all occurrences of the predicate. Consider the inductive definition

inductive $uspec_2$ *:: "nat ⇒ bool"* **where** *"*$uspec_2$ *i ⟹* $uspec_2$ *i"*

Although the rule is trivially valid it cannot be used to prove that $uspec_2$ is *True* for any argument value. Therefore its value is *False* for all arguments.

Monotonicity Properties

Actually, this way of implicit specification when the predicate value is `False` is only possible if all assumptions Q_{ij} used in the rules satisfy a "monotonicity" property for the defined predicate. HOL is able to prove these properties for most forms of the assumptions Q_{ij} automatically and then prove the rules themselves and additional rules to be facts. In the interactive editor these proof activities are displayed in the Output panel.

A common case where a rule assumption Q_{ij} does not satisfy the monotonicity property is if it contains an application of the defined predicate `name` in negated form. Consider the inductive definition

inductive $uspec_3$ :: "nat ⇒ bool" **where** "¬ $uspec_3$ i ⟹ $uspec_3$ i"

Although if $uspec_3$ is defined as $\lambda i.$ `True` it would satisfy the defining rule, the rule cannot be used to prove that $uspec_3$ is `True` for any argument by applying the rule in backward reasoning steps. Isabelle will signal an error for the inductive definition, stating that the monotonicity proof failed for the assumption ¬ $uspec_3$ i. Note that negation may also occur in other syntactic forms like $uspec_3$ i = False or $uspec_3$ i ⟹ i > 0.

3.7.2 Fixed Arguments

It may be the case that one or more arguments of the defined predicate are "fixed", i.e. in all defining rules the values of these arguments in the conclusion are the same as in all assumptions. If this is the case for the first `m` arguments the inductive definition can be specified in the form

inductive name :: "t_1 ⇒ ... ⇒ t_k ⇒ bool"
for y_1 ... y_m
where P_1 | ... | P_n

The variables y_i may be grouped by `and` and types may be specified for (some of) the groups.

Then the defining rules P_i must have the form

$\bigwedge x_{i1} \ldots x_{ip_i}.\ [\![Q_{i1}; \ldots; Q_{iq_i}]\!] \Longrightarrow name\ y_1 \ldots y_m\ term_{i(m+1)} \ldots term_{ik}$

and every occurrence of `name` in the Q_{ij} must also have y_1 ... y_m as its first `m` arguments.

Specifying fixed arguments is optional, it does not have any effect on the defined predicate, however it makes the rules provided by HOL about the predicate simpler (see below).

As an example consider the inductive definition of a divides predicate

inductive *divides :: "nat ⇒ nat ⇒ bool"*
for *m*
where *"divides m 0" | "divides m n ⟹ divides m (n+m)"*

3.7.3 The *cases* Rule

The general form of inductive definition constructs and automatically proves the additional rule

$\llbracket$*name ?a*$_1$ *... ?a*$_k$*; RA*$_1$*; ...; RA*$_n$$\rrbracket$ ⟹ *?P*

where every *RA*$_i$ has the form

$\bigwedge$ *x*$_{i1}$ *... x*$_{ipi}$*.* $\llbracket$*?a*$_1$*=term*$_{i1}$*; ...; ?a*$_k$*=term*$_{ik}$*; Q*$_{i1}$*; ...; Q*$_{iqi}$$\rrbracket$ ⟹ *?P*

and names it *name.cases*.

This rule has the form of an elimination rule for the predicate as described in Section 2.4.4. The major premise is the application *name ?a*$_1$ *... ?a*$_k$ of the defined predicate to arbitrary arguments. When the rule is applied by the methods *erule* or *cases* to a goal it removes a predicate application from the goal assumptions or the input facts, respectively, and splits the goal into cases according to the defining rules of the predicate. The named cases created by the *cases* method are named by numbers starting with 1.

The *cases* rule for the evenness example is *evn.cases*:

$\llbracket$*evn ?a; ?a = 0 ⟹ ?P;* $\bigwedge$*n.* $\llbracket$*?a = n + 2; evn n*$\rrbracket$ ⟹ *?P*$\rrbracket$ ⟹ *?P*

Since it is a case rule it can be displayed by **print_statement** in the alternative form (see Section 2.4.3):

fixes a$_1$ ... a$_k$
assumes *"name* a$_1$ ... a$_k$*"*
obtains *C*$_1$ | ... | *C*$_n$

where every case *C*$_i$ has the form

x$_{i1}$ ... x$_{ipi}$ **where** "a$_1$=*term*$_{i1}$" ... "a$_k$=*term*$_{ik}$" "*Q*$_{i1}$" ... "*Q*$_{iqi}$"

Together with *name.cases* the similar rule *name.simps* is provided. It is an equation which substitutes an arbitrary application of *name* by a disjunction of the cases according to the defining *P*$_i$ and may be used by the simplifier (see Section 2.3.6). It is not added to the simpset automatically, because its application may not terminate (e.g., for *uspec*$_2$ as below).

Effect of Fixed Arguments

If the first *m* predicate arguments are fixed by *for* $y_1 \ldots y_m$ as above, *name.cases* has the simpler form

$\llbracket$*name* $?y_1 \ldots ?y_m$ $?a_1 \ldots ?a_{(k-m)}$; RA_1; ...; $RA_n\rrbracket \Longrightarrow ?P$

where every RA_i has the form

$\bigwedge x_{i1} \ldots x_{ipi}$.
$\llbracket ?a_1$=$term_{i(m+1)}$; ...; $?a_{(k-m)}$=$term_{ik}$; Q_{i1}'; ...; Q_{iqi}'$\rrbracket \Longrightarrow ?P$

and in the Q_{ij}' all applications of *name* have the unknowns $?y_1 \ldots ?y_m$ as their first arguments. In other words, the unknowns from the major premise are directly used in place of the first *m* arguments in all predicate applications without the need of corresponding equations. Therefore the rule *divides.cases* for the divides predicate above is

$\llbracket$*dvds ?m ?a; ?a = 0* $\Longrightarrow$ *?P*; $\bigwedge$*n*. $\llbracket$*?a = n + ?m; dvds ?m n*$\rrbracket \Longrightarrow$ *?P*$\rrbracket$
$\Longrightarrow$ *?P*

Using the *cases* Rule

Due to its form the rule *name.cases* includes information about the implicit definition of *False* predicate values. If a predicate application does not unify with the conclusion of any P_i there will be atleast one assumption $?a_j$=$term_{ij}$ in every RA_i which is false and thus makes RA_i solved as a goal. Together that proves that the predicate application implies arbitrary propositions which means that it is *False*.
Consider the predicate $uspec_1$ defined as above. The rule $uspec_1$*.cases* is

$\llbracket uspec_1$ *?a; ?a=3* $\Longrightarrow$ *?P; ?a=4* $\Longrightarrow$ *?P*$\rrbracket \Longrightarrow$ *?P*

The proposition $uspec_1$ *0 = False* is equivalent to $uspec_1$ *0* $\Longrightarrow$ *False* (because the other direction is trivially valid) which can be proved as follows:

theorem "$uspec_1$ *0* $\Longrightarrow$ *False*"
  apply (*erule* $uspec_1$*.cases*)
  by *simp_all*

The application of $uspec_1$*.cases* creates the goals *0=3* $\Longrightarrow$ *False* and *0=4* $\Longrightarrow$ *False* which are solved by the simplifier.
Note that this does not work for the predicate $uspec_2$. For it $uspec_2$*.cases* is

$\llbracket uspec_2$ *?a*; $\bigwedge$*i*. $\llbracket$*?a = i*; $uspec_2$ *i*$\rrbracket \Longrightarrow$ *?P*$\rrbracket \Longrightarrow$ *?P* and the application by *erule* to uspec_2 *0* $\Longrightarrow$ *False* yields the goal $\bigwedge$*i.* ⟦*0 = i; uspec*$_2$ *i*⟧ $\Longrightarrow$ *False* which is equivalent to the original goal. Therefore even an iterated application by *erule* will not solve the goal in a finite number of steps.

The *cases* Rule in Structured Proofs

The rule *name.cases* is attributed by *[consumes 1]*, so that it can be applied by the *cases* method in structured proofs (see Section 2.4.4). The example above can be written

theorem *"False"* **if** *"uspec*$_1$ *0"*
  using *that*
proof *(cases rule: uspec*$_1$*.cases)* **qed** *simp_all*

Note the use of the structured theorem form to put *uspec*$_1$ *0* into the proof context with name *that* so that it is input by **using** *that* into the structured proof and its initial method *cases* which consumes it.

Moreover, the rule *name.cases* is associated to the defined predicate *name* in the following way: if the first input fact to the *cases* method has an inductively defined predicate *name* as its outermost function, the rule *name.cases* is the default rule applied by the method if no rule is explicitly specified. So the proof for *"False" if "uspec*$_1$ *0"* above can be abbreviated to

proof *cases* **qed** *simp_all*

or even shorter (see Section 2.2.3):

by *cases simp_all*

Note that this only works if the consumed predicate application is taken as input fact, not if it is a goal assumption as in the proof script above.

3.7.4 The Induction Rule

The general form of inductive definition also constructs and automatically proves the additional rule

⟦*name ?a*$_1$ ... *?a*$_k$*; RA*$_1$*; ...; RA*$_n$⟧ $\Longrightarrow$ *?P ?a*$_1$ ... *?a*$_k$

where every *RA*$_i$ has the form

$\bigwedge$ *x*$_{i1}$ ... *x*$_{ipi}$*.* ⟦*Q*$_{i1}$*'; ...; Q*$_{iqi}$*'*⟧ $\Longrightarrow$ *?P term*$_{i1}$ ... *term*$_{ik}$

and in the *Q*$_{ij}$*'* every application *name t*$_1$ ... *t*$_k$ of the predicate to argument terms is replaced by the conjunction *name t*$_1$ ... *t*$_k$ $\wedge$ *?P t*$_1$ ... *t*$_k$. The rule is named *name.induct*.

This rule has the form of an induction rule extended by elimination for the predicate as described in Section 2.4.5. The major premise is the application *name* $?a_1 \ldots ?a_k$ of the defined predicate to arbitrary arguments. The rule is attributed by *[consumes 1]*. When it is applied by the methods *induct* or *induction* to a goal it consumes a predicate application from the input facts or the goal assumptions and splits the goal into cases according to the defining rules of the predicate. The named cases created by the induction methods are named by numbers starting with 1.

The induction rule for the evenness example is *evn.induct*:

$[\![evn\ ?a;\ ?P\ 0;\ \bigwedge n.\ [\![evn\ n;\ ?P\ n]\!] \Longrightarrow ?P\ (n + 2)]\!] \Longrightarrow ?P\ ?a$

Using the Induction Rule

The induction rule can be used to prove properties about the defined predicate *name* which may involve an iterated application of the defining rules. It abstracts the goal conclusion to a function with the same arguments as *name* and then splits it together with the predicate according to the predicate's defining rules. See the Isabelle documentation for corresponding proof techniques.

Like *name.cases* the rule *name.induct* includes information about the implicit definition of *False* predicate values. It covers the cases where a predicate application unifies with the conclusion of a P_i but cannot be reduced in a finite number of backwards reasoning steps.

Consider the predicate $uspec_2$ defined as above. The rule $uspec_2.induct$ is

$[\![uspec_2\ ?x;\ \bigwedge i.\ [\![uspec_2\ i;\ ?P\ i]\!] \Longrightarrow ?P\ i]\!] \Longrightarrow ?P\ ?x$

The proposition $uspec_2\ 0 \Longrightarrow False$ can be proved as follows:

theorem "$uspec_2\ 0 \Longrightarrow False$"
  apply *(induction rule:* $uspec_2$*.induct)*
  by *simp*

The application of $uspec_2.induct$ creates the goal $\bigwedge i.\ [\![uspec_2\ i;\ False]\!] \Longrightarrow False$ which is solved by the simplifier. Actually, this works also for the proposition $uspec_2\ i \Longrightarrow False$ because the conclusion does not depend on the argument of $uspec_2$

Note that this does not work for the predicate $uspec_1$. For it $uspec_1.induct$ is

$[\![uspec_1\ ?x;\ ?P\ 3;\ ?P\ 4]\!] \Longrightarrow ?P\ ?x$

and the application by *induction* to $uspec_1\ 0 \Longrightarrow False$ yields two goals *False* which cannot be proved.

The Induction Rule in Structured Proofs

Like `name.cases` the rule `name.induct` can also be applied in structured proofs (see Section 2.4.5). The example above can be written

```
theorem "False" if "uspec₂ 0"
  using that
proof (induct rule: uspec₂.induct) qed simp
```

Moreover, the rule `name.induct` is associated to the defined predicate `name` in the following way: if the first input fact to the `induction` or `induct` method has an inductively defined predicate `name` as its outermost function, the rule `name.induct` is the default rule applied by the methods if no rule is explicitly specified. So the proof for `"False" if "uspec₂ 0"` above can be abbreviated to

```
proof induction qed simp
```

or even shorter:

```
by induction simp
```

3.7.5 Single-Step Inductive Definitions

A simple case is an inductive definition where no rule assumption Q_{ij} contains an application of the defined predicate `name`. Then for every defining rule a single backward reasoning step will remove the predicate and determine its value. The rule `name.cases` contains the complete information about the defined predicate and is sufficient for proving arbitrary properties about it.

Every predicate defined by an Isabelle definition (see Section 2.1.4)

```
definition name :: "t₁ ⇒ ... ⇒ tₖ ⇒ bool"
  where "name x₁ ... xₙ ≡ term"
```

is equivalent to the predicate defined by the inductive definition

```
inductive name :: "t₁ ⇒ ... ⇒ tₖ ⇒ bool"
  where "term ⟹ name x₁ ... xₙ"
```

Thus inductive definitions are a generalization of the basic Isabelle definitions for predicates.

Here `name` cannot occur in `term` because Isabelle definitions do not support recursion, therefore it is a single-step inductive definition with only one rule. Vice versa, everey single-step inductive definition can be converted to an Isabelle definition, where the `term` is mainly a disjunction of the left sides of all defining rules. Actually, for this case the rule `name.simps` (see above) is equivalent to the defining equation of the corresponding Isabelle definition. For such predicates it is often simpler to use an Isabelle definition. However, if there are a lot of alternative cases in `term` it may be easier to use the form of an inductive definition. Note also, that there are predicates which cannot be defined by an Isabelle definition but can be defined by a (non-single-step) inductive definition.

3.7.6 Mutually Inductive Definitions

Inductively defined predicates may depend on each other. Then they must be defined by a common inductive definition of the extended form

inductive name_1 ... name_m
where P_1 | ... | P_n

The name_i may be grouped by `and` and types may be specified for (some of) the groups.

The defining rules P_i have the same form as described above, however, every conclusion may be an application of one of the defined predicates name_i (with arbitrary ordering of the rules) and every rule assumption Q_{ij} may contain applications of all defined predicates.

The following example defines predicates for even and odd numbers in a mutually inductive way:

```
inductive evn odd :: "nat ⇒ bool" where
  "evn(0)" | "odd(n) ⟹ evn(n+1)" | "evn(n) ⟹ odd(n+1)"
```

The set of defining rules is named name_1_..._name_m`.intros`. For every name_i separate rules name_i`.cases` and name_i`.simps` are created which cover only the defining rules with name_i in the conclusion. As induction rules the set name_1_..._name_m`.inducts` is created containing for every name_i an induction rule with an application of name_i as major premise. Additionally, a rule name_1_..._name_m`.induct` is generated without elimination where the conclusion is an explicit case distinction for all defined predicates.

For the example the rule `evn_odd.induct` is

```
⟦?P1 0; ⋀n. ⟦odd n; ?P2 n⟧ ⟹ ?P1 (n + 1);
  ⋀n. ⟦evn n; ?P1 n⟧ ⟹ ?P2 (n + 1)⟧
⟹ (evn ?x1 ⟶ ?P1 ?x1) ∧ (odd ?x2 ⟶ ?P2 ?x2)
```

3.8 Well-Founded Relations

A binary relation `r` of type `t ⇒ t ⇒ bool` (i.e., the related values are of the same type) is called "well-founded" if every non-empty set of values of type `t` has a "leftmost" element `x` for `r` which means that there is no `y` so that `r y x`. If the relation `(<)` (see Section 3.2.2) is well-founded for a type `t` this is usually described by "every non-empty set has a minimal element".

HOL provides the predicate

```
wfP :: ('a ⇒ 'a ⇒ bool) ⇒ bool
```

which tests an arbitrary binary relation for being well-founded. The predicate

```
wf :: ('a × 'a) set ⇒ bool
```

does the same for a binary relation in tuple set form (see Section 3.6.2).

For a well-founded relation of type `t ⇒ t ⇒ bool` also the universal set `UNIV :: t set` (see Section 5.4.1) has a leftmost element, which is the leftmost value of type `t`. Moreover, from that value all other values of type `t` can be reached in a finite number of steps to a related value, or, if the relation is transitive, even by a single step.

The best-known well-founded relation is the strict less-order `(<)` on the natural numbers. Note that `(≤)` is not well-founded because for every number `n` there is `n` itself so that `n ≤ n`.

3.8.1 Induction

For every well-founded relation `r` the following principle of (transfinite) induction is valid: If a property holds for a value whenever it holds for all values "left" to it, the property holds for all values.

HOL provides the induction rules with elimination (see Section 2.4.5)

```
wfP_induct_rule:
  ⟦wfP ?r; ⋀x. (⋀y. ?r y x ⟹ ?P y) ⟹ ?P x⟧ ⟹ ?P ?a

wf_induct_rule:
  ⟦wf ?r; ⋀x. (⋀y. (y, x) ∈ ?r ⟹ ?P y) ⟹ ?P x⟧ ⟹ ?P ?a
```

Their major premise is well-foundedness of the relation `?r`. The single case corresponds to the induction principle.

3.8.2 The Accessible Part of a Relation

For an arbitrary binary relation `r` of type `t ⇒ t ⇒ bool` generally not all values of `t` can be reached in a finite number of relation steps from a leftmost element. HOL defines the predicate `Wellfounded.accp` by the inductive definition (see Section 3.7)

```
inductive accp :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool"
  for r :: "('a ⇒ 'a ⇒ bool)"
  where "(⋀y. r y x ⟹ accp r y) ⟹ accp r x"
```

Its partial application `accp r` to a binary relation `r` is the predicate which is `True` for all values of `t` which can be reached in a finite number of relation steps from a leftmost element. These values are called the "accessible part" of relation `r`.

HOL also defines the equivalent set-valued function (see Section 3.1.2) for relations represented as tuple sets (see Section 3.6.2)

```
acc :: "('a × 'a) set ⇒ 'a set"
```

which returns the accessible part of a relation as a set.

A binary relation `r` of type `t ⇒ t ⇒ bool` is well-founded if and only if its accessible part `accp r` covers all values of `t`. If `r` is not well-founded there may be several leftmost values in `t` or no such value (in the latter case the accessible part is empty).

For arbitrary binary relations the induction principle can be used on the accessible part. The corresponding induction rules provided by HOL are

```
accp_induct_rule:
 ⟦Wellfounded.accp ?r ?a;
  ⋀x. ⟦Wellfounded.accp ?r x; ⋀y. ?r y x ⟹ ?P y⟧ ⟹ ?P x⟧
 ⟹ ?P ?a

acc_induct_rule:
 ⟦?a ∈ Wellfounded.acc ?r;
  ⋀x. ⟦x ∈ Wellfounded.acc ?r; ⋀y. (y, x) ∈ ?r ⟹ ?P y⟧ ⟹ ?P x⟧
 ⟹ ?P ?a
```

Here the major premise is that the value `?a` for which the property shall be proved belongs to the accessible part of the relation. Also the induction step is only done for values in the accessible part (if a value is in the accessible part then also all values "left" to it).

3.8.3 Measure Functions

A binary relation `r` of type `t ⇒ t ⇒ bool` is well-founded if it is mapped by a function `f :: t ⇒ u` into a well-founded relation `s` on type `u`. This is the case if `r x y ⟹ s (f x) (f y)` for all values `x` and `y` of type `t`.

This property is often used to prove well-foundedness for a binary relation `r` by specifically mapping it into the well-founded order `(<)` on type `nat`. In this context the mapping function `f :: t ⇒ nat` is called a "measure function".

HOL provides the polymorphic function

```
measure :: ('a ⇒ nat) ⇒ ('a × 'a) set
```

which turns a measure function for values of an arbitrary type `'a` to a relation on `'a` in tuple set form. The relation `measure f` relates values `x` and `y` if `(f x) < (f y)`.

The usual way to prove that a relation `r` of type `t ⇒ t ⇒ bool` is well-founded is to design a measure function `f :: t ⇒ nat` and prove that `r x y ⟹ (measure f) x y`.

3.8.4 The *size* Function

HOL introduces the polymorphic function

```
size :: 'a ⇒ nat
```

which is overloaded for many HOL types. If it is not supported for a type an application to a value of that type will result in an error message “No type arity ...”.

The `size` function is intended as a standard measure function for proving well-foundedness of relations on type `'a` with the help of the function `measure`. However, there may be relations where this does not work and a different measure function is required.

Note that the concept of `size` and `measure` actually depends on the specific HOL type `nat`, which is introduced in Section 5.3.

3.9 Recursive Functions

HOL supports the definition of recursive functions in the form

function *name* :: "t_1 ⇒ ... ⇒ t_k ⇒ *type*"
where eq_1 | ... | eq_n ⟨*proof*⟩

The type specification for the function may be omitted if it can be derived from the use of the function in the eq_i.

The definition resembles an Isabelle definition as described in Section 2.1.4, but instead of a single defining equation there may be arbitrary many defining equations eq_i followed by a proof. Also other than for an Isabelle definition the part `name` ... **where** cannot be omitted, because the `name` must be known to check whether the equations have the correct form.

Compared with the inductive definitions in Section 3.7 recursive functions can not only be predicates but may have an arbitrary result type and equations are used instead of defining rules.

3.9.1 The Defining Equations

Each of the defining equations eq_i has the general form of a derivation rule (see Section 2.1.6) with an equation as its conclusion:

$$\bigwedge x_{i1} \ldots x_{ipi}.\ [\![Q_{i1};\ \ldots;\ Q_{iqi}]\!] \Longrightarrow name\ term_{i1} \ldots term_{ik} = term_i$$

It is specified in inner syntax. The separating bars | belong to the outer syntax, therefore each equation must be separately quoted. Note that the conclusion must be an equation using the symbol = instead of ≡. The left side of the equation is restricted to the form of a (non-partial) application of the defined function to argument terms, i.e., if the defined function has k arguments according to its type, in every equation it must be applied to k terms. If the conclusion is no equation or if the left side has a different form an error is signaled.

Other than in an Isabelle definition the equations may be recursive, i.e. the defined function `name` may be used in $term_i$ on the right side of the equation (but not in the Q_{ij} or $term_{ij}$). It may be used in arbitrary ways, also by partial application or by passing it as argument to other functions.

The familiar example of the faculty function can be defined using two defining equations in the form

```
function fac :: "nat ⇒ nat" where
  "fac 0 = 1"
| "⋀n. n > 0 ⟹ fac n = n * fac (n-1)"
⟨proof⟩
```

The bound variables $x_1, \ldots, x_{ipi}$ may occur in the assumptions and in the conclusion. However, if a bound variable occurs in $term_i$ on the right side it must also occur in one or more of the $term_{ij}$ on the left side, otherwise an error is signaled.

After termination has been proved (see the subsection on termination below) for the recursive function definition the defining equations are available as the fact set `name.simps`. Note that no defining equation `name_def` (see Section 2.1.7) exists for recursively defined functions, instead the set `name.simps` plays the corresponding role.

Alternative Rule Forms

As for other derivation rules on theory level (see Section 2.1.6) the explicit bindings are optional, variables occurring free in the assumptions or the conclusion are always automatically bound. As usual, types may be specified for (some of) the variables, so explicit bindings can be used as a central place for specifying types for the variables, if necessary. An equivalent definition for the faculty is

```
function fac :: "nat ⇒ nat" where
  "fac 0 = 1"
| "n > 0 ⟹ fac n = n * fac (n-1)"
⟨proof⟩
```

where the binding of `n` is omitted.

Alternatively an equation eq_i may be specified in the structured form described for derivation rules in Section 2.1.6:

"name $term_{i1}$ ... $term_{ik}$ = $term_i$" **if** "Q_{i1}" ... "Q_{iqi}" **for** x_{i1} ... x_{ipi}

where the assumptions and variables may be grouped by `and` and types may be specified for (some of) the variable groups, however, no names may be specified for assumption groups.

In this form the faculty definition becomes

```
function fac :: "nat ⇒ nat" where
  "fac 0 = 1"
| "fac n = n * fac (n-1)" if "n > 0"
⟨proof⟩
```

Note that on the left side of the equations the arguments may be specified by arbitrary terms, not only by variables. Therefore the faculty function can also be defined in the form

```
function fac2 :: "nat ⇒ nat" where
  "fac2 0 = 1"
| "fac2 (n+1) = (n+1) * fac2 n"
⟨proof⟩
```

where the assumption is not required anymore for the second equation.

It is also possible to explicitly name (some of) the single equations by specifying the recursive definition in the form

function name :: "t_1 ⇒ ... ⇒ t_k ⇒ type" **where**
  $eqname_1$: eq_1 | ... | $eqname_n$: eq_n ⟨proof⟩

Every function `name` defined by the Isabelle definition (see Section 2.1.4)

definition name :: "t_1 ⇒ ... ⇒ t_k ⇒ type"
  where "name x_1 ... x_n ≡ term"

is equivalent to the function defined by the recursive definition

function name :: "t_1 ⇒ ... ⇒ t_k ⇒ type"
  where name_def: "name x_1 ... x_n = term" ⟨proof⟩

which actually does not use recursion.

3.9.2 Covering All Arguments

As described in Section 2.1.3 functions in Isabelle must be total, they must be defined for all possible arguments. It is easy to fail doing so when specifying the defining equations for a recursive function definition. Consider

```
function nototal :: "nat ⇒ nat" where
  "nototal 5 = 6" ⟨proof⟩
```

The function is only defined for the value `5`, no definition is given for the other natural numbers.

Therefore HOL expects a proof that the equations are complete in the sense that they cover all possible cases for the function arguments. In the general case it creates a goal of the form

```
⋀P x.
  ⟦⋀x₁₁ ... x₁ₚ₁. ⟦Q₁₁; ...; Q₁q₁; x = (term₁₁, ..., term₁ₖ)⟧ ⟹ P;
     ...
   ⋀xₙ₁ ... xₙₚₙ. ⟦Qₙ₁; ...; Qₙqₙ; x = (termₙ₁, ..., termₙₖ)⟧ ⟹ P⟧
  ⟹ P
```

which must be proved in the recursive function definition's ⟨*proof*⟩. This goal has the form of a case rule (see Section 2.4.3) before replacing the variable `P` by an unknown upon turning it to a fact. It specifies that `x` covers all possible cases. The variable `x` is set to the tuples of all the argument terms used on the left side of all equations.

Using tuples here depends on the specific HOL type constructor `prod`, which is introduced in Section 5.6. The goal could also be constructed without using tuples, however, it is more compact like that.

Note that the goal does not mention the defined function `name` at all. It is only about the groups of the argument terms in the equations.

For the faculty function defined as above the goal is

```
⋀P x. ⟦x = 0 ⟹ P; ⋀n. ⟦0 < n; x = n⟧ ⟹ P⟧ ⟹ P
```

It can be proved by method `auto` (see Section 2.3.7).

The Proof Method `atomize_elim`

HOL provides the proof method

```
atomize_elim
```

which can be useful for the proof of such goals. It converts the goal to the form

```
⋀x.
    (∃ x₁₁ ... x₁ₚ₁. Q₁₁ ∧ ... ∧ Q₁q₁ ∧ x = (term₁₁, ..., term₁ₖ)
  ∨ ...
  ∨ (∃ xₙ₁ ... xₙₚₙ. Qₙ₁ ∧ ... ∧ Qₙqₙ ∧ x = (termₙ₁, ..., termₙₖ)
```

which directly expresses that the cases together cover all possibilities. It does not contain the technical variable *P* any more and uses the boolean operations and quantifiers introduced by HOL for the type *bool* (see Section 5.1.3). Of course this method can also be used in the proof of other case rules. In many simple cases of recursive function definitions, however, the method is not necessary and the goal can be proved by an automatic method like *auto* or *blast* (see Section 2.3.7).

After the proof of the recursive function definition the goal is available as a case rule named *name.cases*. If applied by the proof method *cases* (see Section 2.4.3) it splits a goal and introduces named cases (named by numbers starting at 1) according to the argument regions in the defining equations of function *name*.

If the function *name* has only one argument *name.cases* is a usual case rule for the argument type, otherwise it is a case rule for the type of the argument tuples.

Recursive Definitions vs. Inductive Definitions

In principle an inductively defined predicate *name* (see Section 3.7) could be defined recursively by adding the cases where it has the value *False*, so that all arguments are covered. The evenness predicate from Section 3.7.1 can be defined by

```
function evn :: "nat ⇒ bool" where
  "evn 0 = True" | "evn 1 = False" | "evn (n+2) = evn n"
⟨proof⟩
```

because the cases *0*, *1*, and *n+2* cover all natural numbers.

However, this is not always possible. It may not be possible to give an explicit specification for the arguments where the predicate value is *False* or the termination proof (see Section 3.9.6) may fail.

Generally, an inductive definition is simpler because it only specifies the positive cases and only these must be proved in proofs of properties of the defined predicate. On the other hand, if the negative cases are of interest they are directly provided by a recursive definition and the defining equations of a recursive definition can be used for rewriting (see Section 2.3.6).

3.9.3 Uniqueness

As usual, a function must be unique, mapping every argument to only one value. It is easy to fail doing so, consider

```
function nounique :: "nat ⇒ nat" where
  "nounique 5 = 6"
| "nounique 5 = 7"
⟨proof⟩
```

The function maps the value `5` to two different values `6` and `7` (and thus is no function anymore).

Therefore HOL expects a proof that the equations are compatible in the sense that they cover disjoint arguments spaces or, if the spaces of two equations overlap, the equations specify the same function values on the overlapping regions. In the general case HOL creates (after the goal for equation completeness) for every pair of equations eq_i and eq_j where $i \le j$ a goal of the form

$$\bigwedge x_{i1} \ \ldots \ x_{ipi} \ x_{j1}' \ \ldots \ x_{jpj}'.$$
$$(term_{i1}, \ \ldots, \ term_{ik}) = (term_{j1}, \ \ldots, \ term_{jk}) \Longrightarrow term_i = term_j$$

Here x_{jb}' is a renamed x_{jb} to avoid a name clash with an x_{ia} if necessary. The renamed variables are consistently replaced in all terms. Moreover, all occurrences of the defined function `name` in $term_i$ and $term_j$ are replaced by `name_sumC` which is the uncurried (see Section 5.6.3) form of `name` where the arguments are specified as a tuple.

Together these are $\sum_{i=1}^{n} i = (n+1) * n/2$ goals where n is the number of equations. The proof of these goals depends on the types and functions used in the equations, for many simple cases it can be done by method `auto` which solves all goals together.

For the faculty function defined as above these are the three goals

```
0 = 0 ⟹ 1 = 1
⋀n. ⟦0 < n; 0 = n⟧ ⟹ 1 = n * fac_sumC (n - 1)
⋀n na. ⟦0 < n; 0 < na; n = na⟧ ⟹ n * fac_sumC (n - 1) = na * fac_sumC
(na - 1)
```

where the first and third are trivial and the second is valid because the two assumptions are a contradiction (because the argument spaces are disjoint). All three goals are solved together by a single application of method `auto` (see Section 2.3.7).

3.9.4 The Domain Predicate

Even if the defining equations cover all possible arguments and define the function values in a unique way the function value may still be underspecified for some arguments. Consider the recursive definition

```
function uspec :: "nat ⇒ nat" where
  "uspec i = uspec i"
  by auto
```

The proof of equation completeness and compatibility is successfully done by the `auto` method and the definition introduces the total function `uspec`, however, no information is available about its result values.

Information about result values must either be specified directly on the right side of a non-recursive equation eq_i or it must be derivable by a finite number of substitutions using the equations. Such a substitution mainly replaces the function arguments specified on the left side of the equation by the arguments used in the recursive calls on the right side.

Therefore HOL creates for every recursive definition of a function `name` the relation `name_rel` which relates for all defining equations the arguments of every recursive call on the right side to the arguments on the left side. Actually, `name_rel` relates argument tuples, like the arguments of `name_sumC` as described above. It is defined inductively (see Section 3.7) using defining rules of the form

$$\bigwedge x_{i1} \dots x_{ipi}.\ [\![Q_{i1}; \dots; Q_{iqi}]\!] \Longrightarrow$$
$$\quad name\_rel\ (term_{ij1}, \dots, term_{ijk})\ (term_{i1}, \dots, term_{ik})$$

where $term_{ijl}$ is the l-th argument in the j-th recursive call of `name` in $term_i$. There is one such rule for every recursive call occurring in the defining equations eq_i. The defined relation `name_rel` never occurs in the Q_{ij}, therefore it is a single-step inductive definition (see Section 3.7.5) which could also have been specified as an Isabelle definition.

For the faculty function defined as above the corresponding definition of the relation `fac_rel` is

```
inductive fac_rel :: "nat ⇒ nat ⇒ bool"
where "0 < n ⟹ fac_rel (n - 1) n"
```

with only one rule because there is only one recursive call in the definition. This relation relates every natural number with its immediate successor.

Now it is possible to characterize the arguments for which the function value is fully specified by the accessible part `Wellfounded.accp name_rel` of `name_rel` (see Section3.8.2). Every substitution by an equation eq_i corresponds to an application of `name_rel` from right to left, every chain of such applications is finite on its accessible part.

For every recursive definition of a function $name :: t_1 \Rightarrow \dots \Rightarrow t_k \Rightarrow type$ HOL defines the abbreviation

```
abbreviation name_dom :: "(t₁ × ... × tₖ) ⇒ bool" where
  "name_dom ≡ Wellfounded.accp name_rel"
```

called the “domain predicate” for the argument tuples for which `name` is fully specified.

For the faculty function the domain predicate is defined by

```
abbreviation fac_dom :: "nat ⇒ bool" where
  "fac_dom ≡ Wellfounded.accp fac_rel"
```

Since every natural number can be reached by a finite number of steps starting at `0` this predicate is `True` for all natural numbers.

3.9.5 Rules Provided by Recursive Definitions

HOL automatically creates and proves some additional rules from a recursive definition and its proof. The domain predicate is used in these rules to exclude underspecified cases.

Simplification Rules

HOL automatically creates and proves simplification rules which directly correspond to the defining equations eq_i. Every such rule is guarded by the domain predicate for the arguments of the substituted function application. For the equation eq_i in the general form as given above the rule is

$$\llbracket Q_{i1};\ \ldots;\ Q_{iqi};\ name\_dom\ (term_{i1},\ \ldots,\ term_{ik})\rrbracket \Longrightarrow$$
$$name\ term_{i1}\ \ldots\ term_{ik}\ =\ term_i$$

where all occurrences of the bound variables $x_{i1},\ \ldots,\ x_{ipi}$ have been replaced by corresponding unknowns. The set of all these simplification rules is named `name.psimps`. The rules are not added to the simpset automatically, they can be explicitly used for “unfolding” the recursive definition in one or more steps. If individual names have been specified for (some of) the eq_i these names denote the corresponding facts in `name.psimps`.

Elimination Rule

HOL also creates an elimination rule (see Section 2.4.4) `name.pelims` where the major premise has the form $name\ ?x_1\ \ldots\ ?x_k\ =\ ?y$ and the rule replaces it mainly by case assumptions according to the defining equations.

Induction Rule

HOL also creates and proves a single induction rule `name.pinduct` of the form

$$\llbracket name\_dom\ (?a_1,\ \ldots,\ ?a_k);\ RA_1;\ \ldots;\ RA_n\rrbracket \Longrightarrow ?P\ ?a_1\ \ldots\ ?a_k$$

where every RA_i has the form

$$\bigwedge x_{i1} \ldots x_{ip_i}.\ \llbracket name\_dom\ (term_{i1}, \ldots, term_{ik});\ Q_{i1};\ \ldots;\ Q_{iq_i};\ R_{i1};\ \ldots;\ R_{ir_i}\rrbracket \Longrightarrow\ ?P\ term_{i1} \ldots term_{ik}$$

and every R_{ij} has the form $?P\ term_{ij1} \ldots term_{ijk}$ where $term_{ijl}$ is the l-th argument in the j-th recursive call of *name* in $term_i$.

It can be used for induction with elimination (see Section 2.4.5) where the major premise is the domain predicate for the arguments of the proved property *?P* which is usually a property of the defined function *name* applied to these arguments. The cases correspond to the defining equations and are named by numbers starting with 1. The induction step in every case goes from the arguments of the recursive calls in $term_i$ to the arguments on the left side of eq_i. This form of induction is also called "computation induction".

For the faculty function defined as above the induction rule *fac.pinduct* is

$$\llbracket fac\_dom\ ?a;\ fac\_dom\ 0 \Longrightarrow\ ?P\ 0;\ \bigwedge n.\ \llbracket fac\_dom\ n;\ 0 < n;\ ?P\ (n - 1)\rrbracket \Longrightarrow\ ?P\ n\rrbracket \Longrightarrow\ ?P\ ?a$$

3.9.6 Termination

In programming languages a recursively defined function *name* is only considered correct if its value can be determined for every argument by simplification using *name.psimps* in a finite number of steps (i.e., its computation terminates). This is only the case if the argument (tuple) satisfies the domain predicate. Therefore it is often of interest to prove that this is the case for all possible argument (tuple)s. HOL provides specific support for proving this property.

The corresponding theorem for a recursively defined function $name :: t_1 \Rightarrow \ldots \Rightarrow t_k \Rightarrow type$ is

theorem "$\forall x :: (t_1 \times \ldots \times t_k).\ name\_dom\ x$" $\langle proof\rangle$

(for the quantifier ∀ see Section 5.1.3). HOL provides the equivalent abbreviated form

termination *name* $\langle proof\rangle$

which is called a "termination proof" for the recursive function *name*. If *name* is omitted it refers to the last previously defined recursive function.

The Proof Method *relation*

Note that this theorem is equivalent to the property that the relation `name_rel` is well-founded. As described in Section 3.8.3 it can be proved by proving that `name_rel x y` $\Longrightarrow$ `(measure f) x y` holds for a measure function `f ::` $(t_1 \times \ldots \times t_k) \Rightarrow$ `nat`. HOL provides the proof method

`relation "M"`

which replaces the original goal of a termination proof for `name` by the goals

`wf M`
R_1
...
R_r

where every R_h corresponds to a defining rule for `name_rel` (see Section 3.9.4) in `name_rel.intros` (see Section 3.7.1) and has the form

$$\bigwedge x_{i1} \ldots x_{ipi}.\ [\![Q_{i1}; \ldots; Q_{iqi}]\!] \Longrightarrow$$
$$((term_{ij1}, \ldots, term_{ijk}), (term_{i1}, \ldots, term_{ik})) \in M$$

where $term_{ijl}$ is the l-th argument in the j-th recursive call of `name` in $term_i$. The goals $R_1, \ldots, R_r$ together are equivalent to the goal `name_rel x y` $\Longrightarrow$ `M x y`.
The proof method is usually applied in the form

`relation "measure f"`

for an appropriate measure function `f` as above. It must be constructed so that for every tuple $(term_{ij1}, \ldots, term_{ijk})$ of arguments for a recursive call the value is strictly lower than for the argument tuple $(term_{i1}, \ldots, term_{ik})$ in the defining equations eq_i. The resulting goals `wf (measure f)` and $R_1, \ldots, R_r$ are often solved by method `auto` (see Section 2.3.7). Then the termination proof has the form

termination `name` **by** `(relation "measure f") auto`

For the faculty function defined as above the goal of the termination proof is

$\forall x$ `:: nat. fac_dom x`

which is true as argued above. To prove it, the identity function $\lambda n.\ n$ is an applicable measure function because $n > 0 \Longrightarrow n\text{-}1 < n$. Applying the `relation` method with it yields the two goals

$$wf\ (measure\ (\lambda n.\ n))$$
$$\bigwedge n.\ 0 < n \Longrightarrow (n - 1, n) \in measure\ (\lambda n.\ n)$$

They can be solved together by method `auto`, therefore a termination proof for the faculty definition can be specified as

termination *fac* **by** *(relation "measure (λn. n)") auto*

The Proof Methods *lexicographic_order* and *size_change*

HOL also provides the proof method

lexicographic_order

and the stronger method

size_change

for termination proofs. They try to automatically construct a measure function from the *size* function (see Section 3.8.4) combined in a lexicographic way for argument tuples. They are applied to the original goal and if they are successful they solve it completely, otherwise an error is signaled. Using the first method a termination proof has the form

termination *name* **by** *lexicographic_order*

Since for type *nat* the *size* function is the identity the termination proof for the faculty function may also be specified as

termination *fac* **by** *lexicographic_order*

3.9.7 Rules Provided by Termination Proofs

A termination proof for a function *name* using the command **termination** provides the additional rules *name.simps*, *name.elims*, and *name.induct*. They result from the corresponding rules provided by the recursive definition by removing all applications of the domain predicate *name_dom*. This is valid because the termination proof has shown that it is always *True*.

Specifically, the resulting simplification rules exactly correspond to the defining equations and can be used for unfolding the definition for a function application in one or more steps.

If individual names have been specified for (some of) the eq_i the termination proof replaces the associated facts from *name.psimps* by those from *name.simps*.

The resulting induction rule is a plain induction rule without elimination of the form

$[\![RA_1; \ldots; RA_n]\!] \Longrightarrow ?P\ ?a_1\ \ldots\ ?a_k$

where every RA_i has the form

$\bigwedge$ x_{i1} ... x_{ipi}. $[\![Q_{i1}; \ldots; Q_{iqi}; R_{i1}; \ldots; R_{iri}]\!]$
  $\Longrightarrow$?P $term_{i1}$ ... $term_{ik}$

and every R_{ij} has the form ?P $term_{ij1}$ ... $term_{ijk}$ where $term_{ijl}$ is the *l*-th argument in the *j*-th recursive call of *name* in $term_i$.

For the faculty function defined as above the induction rule *fac.induct* is

$[\![$?P 0; $\bigwedge$n. $[\![$0 < n; ?P (n - 1)$]\!]$ $\Longrightarrow$?P n$]\!]$ $\Longrightarrow$?P ?a

To use the rule with the proof methods *induct* and *induction* (see Section 2.4.5) it must always be specified explicitly in the form

```
induct ... rule: name.induct
```

3.9.8 Mutual Recursion

If several recursive functions are defined depending on each other they must be defined together in a single recursive definition of the form

function $name_1$ ... $name_m$
where eq_1 | ... | eq_n ⟨*proof*⟩

The $name_i$ may be grouped by *and* and types may be specified for (some of) the groups.

The defining equations eq_i have the same form as described above, however, every left side may be an application of one of the defined functions $name_i$ (with arbitrary ordering of the rules) and every right side $term_i$ may contain applications of all defined functions.

A mutual recursive definition of the predicates *evn* and *odd* (see Section 3.7.6) is

```
function evn odd :: "nat ⇒ bool"
where "evn 0 = True" | "odd 0 = False"
| "evn (n+1) = odd n" | "odd (n+1) = evn n"
⟨proof⟩
```

The simplification and elimination rules are provided for every defined function as $name_i$.*psimps*, $name_i$.*pelims*, $name_i$.*simps*, and $name_i$.*elims*. The induction rules are provided for all defined functions together in the sets named $name_1$_..._$name_m$.*pinduct* and $name_1$_..._$name_m$.*induct*.

The domain predicate and the corresponding relation are common for all defined functions and are named $name_1$_..._$name_m$_*dom* and $name_1$_..._$name_m$_*rel*. They are defined on values of the sum type (see Section 5.9) of the argument tuples of all defined functions, this is also the case for the function *name*$_1$*_..._name*$_m$*_sumC* used in the goals for the uniqueness proof. Also a measure function used in a termination proof must be defined on this sum type.

A termination proof can use the name of either defined function to refer to the mutual recursive definition.

Chapter 4

Isabelle HOL Type Definitions

This chapter introduces mechanisms defined by HOL which are used to populate HOL with many of its mathematical objects and functions and which can also be used to extend HOL to additional kinds of objects. Basically these mechanisms support the definition of new types.

4.1 Algebraic Types

Roughly an algebraic type is equivalent to a union of tuples with support for recursion, which allows nested tuples. In this way most data types used in programming languages can be covered, such as records, unions, enumerations, and pointer structures. Therefore HOL also uses the notion "datatype" for algebraic types.

4.1.1 Definition of Algebraic Types

Basically, an algebraic type is defined in the form

datatype *name* = alt_1 | ... | alt_n

where *name* is the name of the new algebraic type and every alternative alt_i is a "constructor specification" of the form

$cname_i$ "$type_{i1}$" ... "$type_{iki}$"

The $cname_i$ are names and the $type_{ij}$ are types. The types are specified in inner syntax and must be quoted, if they are not a single type name. All other parts belong to the outer syntax.

Recursion is supported for the types, i.e., the name *name* of the defined type may occur in the type specifications $type_{ij}$. However, there must be atleast one constructor specification which is not recursive, otherwise the definition does not “terminate”. Isabelle checks this condition and signals an error if it is not satisfied.

As a convention, capitalized names are used in HOL for the *cname*$_i$.

An example for a datatype definition with two constructor specifications is

```
datatype coord =
  Dim2 nat nat
| Dim3 nat nat nat
```

Its value set is equivalent to the union of pairs and triples of natural numbers.

An example for a recursive datatype definition with two constructor specifications is

```
datatype tree =
  Leaf nat
| Tree nat tree tree
```

Its value set is equivalent to the set of all binary trees with a natural number in every node.

Like declared types algebraic types may be parameterized (see Section 2.1.2):

datatype ('name$_1$,...,'name$_m$) name = alt$_1$ | ... | alt$_n$

where the 'name$_i$ are the type parameters. They may occur in the type specifications type$_{ij}$, i.e., the type$_{ij}$ may be polymorphic (see Section 2.1.2). As usual, the parentheses may be omitted if there is only one type parameter.

An example for a parameterized datatype definition with one type parameter is

```
datatype 'a coordx =
  Dim2 'a 'a
| Dim3 'a 'a 'a
```

Its value set is equivalent to the union of pairs and triples of values of the type parameter. The type `coord` is equivalent to the type `nat coordx`. The type `real coordx` is equivalent to the union of pairs and triples of values of type `real` of the real numbers.

4.1.2 Constructors

Every *cname*$_i$ is used by the definition to introduce a “(value) constructor function”, i.e., a constant

cname$_i$ :: "type$_{i1}$ ⇒ ... ⇒ type$_{iki}$ ⇒ name" which is a function with `ki` arguments mapping their arguments to values of the new type `name`.

Every datatype definition constitutes a separate namespace for the functions it introduces. Therefore the same names may be used in constructor specifications of different datatype definitions. If used directly, a name refers to the constructor function of the nearest preceding datatype definition. To refer to constructor functions with the same name of other datatypes the name may be qualified by prefixing it with the type name in the form `name.cname`$_i$.

The definition of type `coord` above introduces the two constructor functions `Dim2 :: nat ⇒ nat ⇒ coord` and `Dim3 :: nat ⇒ nat ⇒ nat ⇒ coord`. Their qualified names are `coord.Dim2` and `coord.Dim3`.

Constructing Values

These constructor functions are assumed to be injective, thus their result values differ if atleast one argument value differs. This implies that the set of all values of the constructor function `cname`$_i$ is equivalent to the tuples of the value sets of `type`$_{i1}$ ... `type`$_{iki}$: for every tuple of arguments there is a constructed value and vice versa. Note, however, that as usual the values of the new type are distinct from the values of all other types, in particular, they are distinct from the argument tuples.

Moreover the result values of different constructor functions are also assumed to be different. Together the set of all values of the defined type is equivalent to the (disjoint) union of the cartesian products of all constructor argument types. Moreover, every value of the type may be denoted by a term

`cname`$_i$ `term`$_1$ ... `term`$_{ki}$

where each `term`$_j$ is of type `type`$_{ij}$ and specifies an argument for the constructor function application.

Together, datatypes have what is called “free constructors” in Isabelle: the constructors are injective, disjoint, and exhaustive (they cover all values of the type).

Values of type `coord` as defined above are denoted by terms such as `Dim2 0 1` and `Dim3 10 5 21`.

Constant Constructors and Enumeration Types

A constructor specification may consist of a single constructor name `cname`$_i$, then the constructor function has no arguments and always constructs the same single value. The constructor is equivalent to a constant of type `name`. As a consequence an “enumeration type” can be defined in the form

datatype `three = Zero | One | Two`

This type *three* has three values denoted by *Zero*, *One*, and *Two*.

Types with a Single Constructor

If a datatype definition consists of a single constructor specification its value set is equivalent to the constructor argument tuples. The corresponding tuples have a separate component for every constructor argument type. As a consequence a "record type" can be defined in the form

datatype *recrd = MkRecrd nat "nat set" bool*

Its values are equivalent to triples where the first component is a natural number, the second component is a set of natural numbers, and the third component is a boolean value. An example value is denoted by *MkRecrd 5 {1,2,3} True*.

Since there must be atleast one nonrecursive constructor specification, definitions with a single constructor specification cannot be recursive.

4.1.3 Destructors

Since constructor functions are injective it is possible to determine for every value of the defined type the value of each constructor argument used to construct it. Corresponding mechanisms are called "destructors", there are three different types of them.

Selectors

The most immediate form of a destructor is a selector function. For the constructor argument specified by type_{ij} the selector function is a function of type $\mathit{name} \Rightarrow \mathit{type}_{ij}$. For every value constructed by $\mathit{cname}_i\ \mathit{term}_1 \ldots \mathit{term}_{ki}$ it returns the value denoted by term_j.

The names of selector functions must be specified explicitly. This is done using the extended form of a constructor specification

cname_i (sname_{i1} : "type_{i1}") ... (sname_{iki} : "type_{iki}")

where the sname_{ij} are the names used for the corresponding selector functions. Selector names may be specified for all or only for some constructor arguments. As for constructors, selector names belong to the namespace of the defined type and may be qualified by prefixing the type name.

An example datatype definition with selectors is

datatype *recrd = MkRecrd (n:nat) (s:"nat set") (b:bool)*

It shows that the selector functions correspond to the field names used in programming languages in record types to access the components. For every term *r* of type *recrd* the selector term *s r* denotes the set component of *r*.

An example for a datatype with multiple constructor specifications is

```
datatype coord =
  Dim2 (x:nat) (y:nat)
| Dim3 (x:nat) (y:nat) (z:nat)
```

Note that the selectors *x* and *y* are specified in both alternatives. Therefore a single selector function $x :: coord \Rightarrow nat$ is defined which yields the first component both for a two-dimensional and a three-dimensional coordinate and analogously for *y*. If instead the definition is specified as

```
datatype coord =
  Dim2 (x2:nat) (y:nat)
| Dim3 (x3:nat) (y:nat) (z:nat)
```

two separate selector functions *x2* and *x3* are defined where the first one is only applicable to two-dimensional coordinates and the second one only to three-dimensional coordinates.

If a selector name does not occur in all constructor specifications, the selector function is still total, like all functions in Isabelle, but it is underspecified (see Section 2.1.3). It maps values constructed by other constructors to a unique value of its result type, even if that other constructor has no argument of this type. However, no information is available about that value.

For the type *coord* the selector function $z :: coord \Rightarrow nat$ is also applicable to two-dimensional coordinates, however, the values it returns for them is not specified.

Such selector values are called "default selector values". They may be specified in the extended form of a datatype definition

datatype $name$ = alt_1 | ... | alt_n
where "$prop_1$" | ... | "$prop_m$"

where every $prop_p$ is a proposition of the form

$sname_{ij}$ ($cname_q$ var_1 ... var_{kq}) = $term_p$

and specifies $term_p$ as the default value of selector $sname_{ij}$ for values constructed by $cname_q$.

The definition

```
datatype coord =
  Dim2 (x:nat) (y:nat)
| Dim3 (x:nat) (y:nat) (z:nat)
where "z (Dim2 a b) = 0"
```

specifies `0` as default value for selector `z` if applied to a two-dimensional coordinate.

Discriminators

If an underspecified selector is applied to a datatype value it may be useful to determine which constructor has been used to construct the value. This is supported by discriminator functions. For every constructor specification for *cname*$_i$ the discriminator function has type `name ⇒ bool` and returns true for all values constructed by *cname*$_i$. Like selector names, discriminator names must be explicitly specified using the extended form of a datatype definition

datatype *name = dname*$_1$*: alt*$_1$ | ... | *dname*$_n$*: alt*$_n$

Discriminator names may be specified for all alternatives or only for some of them. Note that for a datatype with a single constructor the discriminator returns always `True` and for a datatype with two constructors one discriminator is the negation of the other.

An example datatype definition with discriminators is

```
datatype coord =
  is_2dim: Dim2 nat nat
| is_3dim: Dim3 nat nat nat
```

In a datatype definition both discriminators and selectors may be specified.

The *case* Term

Additionally to using discriminators and selectors HOL supports `case` terms. A `case` term specifies depending on a datatype value a separate term variant for every constructor of the datatype. In these variants the constructor arguments are available as bound variables.

A `case` term for a datatype `name` defined as in Section 4.1.1 has the form

case term of
  cname$_1$ *var*$_{11}$ ... *var*$_{1k1}$ ⇒ *term*$_1$
| ...
| *cname*$_n$ *var*$_{n1}$ ... *var*$_{nkn}$ ⇒ *term*$_n$

where `term` is of type `name` and the *term*$_i$ have an arbitrary but common type which is also the type of the `case` term. In the alternative for constructor *cname*$_i$ the *var*$_{11}$ ... *var*$_{1k1}$ must be distinct variables, they are bound to the constructor arguments and may be used in *term*$_i$ to access them. The value of *var*$_{ij}$ is the same as the value selected by *sname*$_{ij}$ *term*.

Actually, a `case` term is only an alternative syntax for the function application term

```
case_name
  (λ var₁₁ ... var₁ₖ₁. term₁)
  ...
  (λ varₙ₁ ... varₙₖₙ. termₙ)
  term
```

Here *case_name* is the "case combinator" function for the datatype *name*. It takes as arguments *n* functions which map the corresponding constructor arguments to the term variant (or the term variant itself if the constructor has no arguments) and the *term* of type *name* as final argument. Note that the constructor names *cname*$_i$ do not occur here, the constructor corresponding to a term variant is only determined by the argument position *i* compared with the position of the constructor in the datatype definition.

If *cv* is a variable or constant of type *coord* an example *case* term for it is

```
case cv of
  Dim2 a b ⇒ a + b
| Dim3 a b c ⇒ a + b + c
```

It denotes the sum of the coordinates of *cv*, irrespective whether *cv* is two-dimensional or three-dimensional. The corresponding case combinator application term is

```
case_coord (λ a b. a+b) (λ a b c. a+b+c) cv
```

A *case* term is useful even for a datatype with a single constructor. If *rv* is of type *recrd* as defined in Section 4.1.3 the *case* term

```
case rv of MkRecrd nv sv bv ⇒ term
```

makes the components of *rv* locally available in *term* as *nv*, *sv*, *bv*. It is equivalent to *term* where *nv*, *sv*, and *bv* have been substituted by the selector applications *(n rv)*, *(s rv)*, and *(b rv)*.

The variant terms in a *case* term cannot be matched directly by a **let** statement in a proof (see Section 2.2.11). The statement

```
let "case rv of MkRecrd nv sv bv ⇒ ?t"
  = "case rv of MkRecrd nv sv bv ⇒ term"
```

will fail to bind *?t* to *term* because then the variables *nv*, *sv*, and *bv* would occur free in it and the relation to the constructor arguments would be lost. Instead, the statement

```
let "case rv of MkRecrd nv sv bv ⇒ ?t nv sv bv"
  = "case rv of MkRecrd nv sv bv ⇒ term"
```

successfully binds *?t* to the lambda term *λnv sv bv. term* which denotes the function which results in *term* when applied to the constructor arguments and occurs as argument of the case combinator function.

4.1.4 Parameterized Algebraic Types as Bounded Natural Functors

As described in Section 4.1.2 every datatype value can be thought of being equivalent to a tuple of constructor argument values, so in some sense the constructor argument values are "contained" in the datatype value. If an algebraic type has type parameters, these may occur as constructor argument types or as parts thereof. In this sense every datatype value is a "container" of a certain number of values of every type parameter.

As an example, every value of the polymorphic datatype `'a coordx` defined in Section 4.1.1 contains two or three values of the type parameter `'a`.

Retrieving Contained Values

More generally, a type constructor `name` is called a "bounded natural functor" (BNF), if it has for every type parameter $'p_i$ a function $('p_1,\ldots,'p_m)$ `name` $\Rightarrow$ $'p_i$ `set` which returns for every value of type $('p_1,\ldots,'p_m)$ `name` the set of contained values of type $'p_i$, and if all these sets are "bounded", i.e. their maximal size is only determined by `name` and not by the actual types substituted for the type parameters.

As an example, values of type `'a coordx` contain maximally `3` values of type `'a`, irrespective whether `'a` is substituted by type `bool`, where there are only two possible values, or by type `nat`, where there are infinitely many possible values. For a recursive datatype the set of contained values is often not bound by a number, instead it is bound to be finite, even if the actual type argument has inifinitely many values.

Every definition

datatype $('p_1,\ldots,'p_m)$ `name = `alt_1` | ... | `alt_n

of a parameterized datatype with type parameters $'p_1,\ldots,'p_m$ introduces for every type parameter $'p_i$ this "set function" as

`seti_name :: "`$('p_1,\ldots,'p_m)$` name ⇒ `$'p_i$` set"`

If `m = 1` the single set function is named `set_name`, if `m = 2` the two set functions are named `set1_name` and `set2_name`.

For the datatype `coordx` the only set function is `set_coordx`. It maps every value of a type `t coordx` to the set of either two or three coordinate values of type `t`.

Replacing Contained Values

Moreover, for a bounded natural functor it must be possible to replace the contained values "in-place" without modifying any other parts of the container value. Contained values are replaced by applying a function to them.

This property can be modeled by a single function called a “map function”. It takes as arguments one function f_i for every type parameter $'p_i$ and returns a function on the container values which replaces every contained value x of type $'p_i$ by $f_i\ x$.

Every datatype definition as above introduces the map function as

```
map_name :: "('p₁ ⇒ 'q₁) ⇒ ... ⇒ ('pₘ ⇒ 'qₘ)
   ⇒ ('p₁,...,'pₘ) name ⇒ ('q₁,...,'qₘ) name"
```

It takes as arguments m functions $f_1, \ldots, f_m$ and a datatype value. Every f_i may map its arguments of type $'p_i$ to values of the same type or of a different type $'q_i$. In the latter case also the resulting datatype value is of a different type (a different instance of the same parameterized datatype).

An alternative way of understanding `map_name` is that the partial application (see Section 2.1.3) $map\_name\ f_1\ \ldots\ f_m$ “lifts” the m functions to a function between instances of type $('p_1,\ldots,'p_m)\ name$ and $('q_1,\ldots,'q_m)\ name$. In particular, if $m=1$ then `map_name` lifts every function $f :: t_1 \Rightarrow t_2$ to the function $(map\_name\ f) :: t_1\ name \Rightarrow t_2\ name$.

The function `map_coordx` has type `('p ⇒ 'q) ⇒ 'p coordx ⇒ 'q coordx`. For instance, if `f :: real ⇒ nat` is the function that rounds every real number to the next natural number, the application `map_coordx f cv` replaces the real coordinates in `cv` of type `real coordx` by rounded natural coordinates, resulting in a value of type `nat coordx`.

In general a type constructor with such a map function is called a “functor”. It does not only support constructing values from values of the parameter types, but also functions from functions on the parameter types.

Constructing Predicates and Relations

A bounded natural functor can use the sets of contained values returned by the set functions to lift predicates and relations (see Section 3.1) in a similar way from contained values to container values. This is modeled by a “predicator function” and a “relator function”.

For a datatype definition as above the predicator function is provided as

```
pred_name :: "('p₁ ⇒ bool) ⇒ ... ⇒ ('pₘ ⇒ bool)
    ⇒ ('p₁,...,'pₘ) name ⇒ bool"
```

It takes as arguments m unary predicates $p_1, \ldots, p_m$ and a datatype value x and tests whether all values in `seti_name x` satisfy the corresponding predicate p_i.

The partial application $pred\_name\ p_1\ \ldots\ p_m$ lifts the predicates $p_1, \ldots, p_m$ to a predicate on the corresponding instance of type $('p_1,\ldots,'p_m)\ name$.

The function *pred_coordx* has type *('p ⇒ bool) ⇒ 'p coordx ⇒ bool*. For instance, if *cv* is of type *nat coordx* the term *pred_coordx (λn. n=0) cv* tests whether all coordinates of *cv* are *0*.

The relator function is provided as

```
rel_name :: "('p$_1$ ⇒ 'q$_1$ ⇒ bool) ⇒ ... ⇒ ('p$_m$ ⇒ 'q$_m$ ⇒ bool)
   ⇒ ('p$_1$,...,'p$_m$) name ⇒ ('q$_1$,...,'q$_m$) name ⇒ bool"
```

It takes as arguments *m* binary relations r_1, ..., r_m and two datatype values *x, y* and tests whether all pairs of values contained at the same position in *x* and *y* are related by the corresponding r_i. This is done by constructing a container of type *('p$_1$×'q$_1$, ..., 'p$_m$×'q$_m$) name* where all contained values are pairs (see Section 3.6) which can be retrieved by the set functions and tested whether they are related. Using the map function every contained pair can be replaced by its first or second component, respectively, resulting in the containers to be tested for being related.

The partial application *rel_name* r_1 ... r_m lifts the relations r_1, ..., r_m to a relation between the corresponding instances of the types *('p$_1$,...,'p$_m$) name* and *('q$_1$,...,'q$_m$) name*.

The function *rel_coordx* has type *('p ⇒ 'q ⇒ bool) ⇒ 'p coordx ⇒ 'q coordx ⇒ bool*. For instance, if cv_1 and cv_2 are of type *nat coordx* the term *rel_coordx (≤)* cv_1 cv_2 tests whether cv_1 and cv_2 have the same dimension and every coordinate in cv_1 is lower or equal to the corresponding coordinate in cv_2.

Specifying Names for the BNF Functions

The form

```
datatype (sname$_1$: 'p$_1$,..., sname$_m$: 'p$_m$) name = alt$_1$ | ... | alt$_n$
  for map: mname pred: pname rel: rname
```

of a datatype definition allows to define alternate names *sname*$_i$, *mname*, *pname*, *rname* for (some of) the set, map, predicator, and relator functions.

4.1.5 Rules

A datatype definition also introduces a large number of named facts about the constructors and destructors of the defined type. All fact names belong to the namespace of the datatype definition. Since the fact names cannot be specified explicitly, all datatype definitions use the same fact names, therefore the fact names must always be qualified by prefixing the type name.

Several rules are configured for automatic application, e.g., they are added to the simpset for automatic application by the simplifier (see Section 2.3.6). Other rules must be explicitly used by referring them by their name.

Only some basic rules are described here, for more information refer to the Isabelle documentation about datatypes.

Simplifier Rules

The rules added by a definition for a datatype *name* to the simpset (see Section 2.3.6) support many ways for the simplifier to process terms with constructors and destructors.

The rule set `name.inject` states that every non-constant constructor is injective, the rules are of the form

$$((cname_i\ ?x_1\ \dots\ ?x_{ki}) = (cname_i\ ?y_1\ \dots\ ?y_{ki})) = (?x_1 = ?y_1 \wedge \dots \wedge ?x_{ki} = ?y_{ki})$$

The rule set `name.distinct` states that values constructed by different constructors are different, the rules are of the form

$$(cname_i\ ?x_1\ \dots\ ?x_{ki}) \neq (cname_j\ ?y_1\ \dots\ ?y_{kj})$$

where $i \neq j$.

The rule set `name.sel` provides “defining equations” for the selectors of the form

$$sname_{ij}\ (cname_i\ ?x_1\ \dots\ ?x_{ki}) = ?x_j$$

The rule set `name.case` provides equations for simplifying `case` terms where the discriminating term is directly specified by a constructor application. They have the form

```
(case (cname_i ?x_1 ... ?x_ki) of
    cname_1 var_11 ... var_1k1 ⇒ ?f_1 var_11 ... var_1k1
  | ...
  | cname_n var_n1 ... var_nkn ⇒ ?f_n var_n1 ... var_nkn)
= ?f_i ?x_1 ... ?x_ki
```

Note that each branch is specified as an application of a function $?f_i$ so that the variables bound in the branch can be substituted by the arguments of the constructor application.

Depending on the datatype definition there may be additional simplifier rules. In particular, if the datatype is parameterized, simplifier rules are generated for the functions described in Section 4.1.4. The set of all rules added to the simpset is named `name.simps`. By displaying it using the **thm** command (see Section 2.1.7) it can be inspected to get an idea how the simplifier processes terms for a specific datatype.

Case Rule

Every definition for a datatype `name` introduces a rule corresponding to the exhaustiveness of the free constructors (see Section 4.1.2). It has the form

```
name.exhaust:
  ⟦⋀x₁ ... xₖ₁. ?y = cname₁ x₁ ... xₖ₁ ⟹ ?P;
   ... ;
   ⋀x₁ ... xₖₙ. ?y = cnameₙ x₁ ... xₖₙ ⟹ ?P⟧ ⟹ ?P
```

According to Section 2.4.3 this rule is a case rule. It is automatically associated with the datatype for use by the `cases` method. Therefore the application of the method

```
cases "term"
```

where `term` is of type `name` splits an arbitrary goal into `n` subgoals where every subgoal uses a different constructor to construct the `term`.

The names for the named contexts created by the `cases` method are simply the constructor names `cname`$_i$. Therefore a structured proof using case based reasoning for a `term` of datatype `name` has the form

```
proof (cases "term")
  case (cname₁ x₁ ... xₖ₁) ... show ?thesis ⟨proof⟩
next
...
next
  case (cnameₙ x₁ ... xₖₙ) ... show ?thesis ⟨proof⟩
qed
```

The names `x`$_i$ of the locally fixed variables can be freely selected, they denote the constructor arguments of the corresponding constructor. Therefore the case specification `(cname`$_i$ `x`$_1$ `...` `x`$_{ki}$`)` looks like a constructor application to variable arguments, although it is actually a context name together with locally fixed variables.

Split Rule

A `case` term (see Section 4.1.3) is only processed automatically be the simplifier, if the discriminating term is a constructor application (see the `name.case` rule set above). Otherwise it is only processed if a corresponding split rule is configured for it (see Section 2.3.6). Every definition for a datatype `name` introduces such a split rule. It has the form

```
name.split:
  ?P(case ?t of
     cname₁ x₁₁ ... x₁ₖ₁ ⇒ ?t₁ x₁₁ ... x₁ₖ₁
   | ...
   | cname_n x_n1 ... x_nkn ⇒ ?t_n x_n1 ... x_nkn) =
  (   (?t = cname_1 x_11 ... x_1k1 ⟶ ?P(?t_1 x_11 ... x_1k1))
   ∧ ...
   ∧ (?t = cname_n x_n1 ... x_nkn ⟶ ?P(?t_n x_n1 ... x_nkn)))
```

As described in Section 2.3.6 the rule splits a goal with a `case` term for type `name` in the conclusion into goals where the `case` term is replaced by the terms in the cases. Note that the sub-terms of the `case` term are specified by unknowns, so the rule unifies with arbitrary `case` terms for type `name`. Also note, that the `?t`$_i$ are specified with arguments, so that they will be matched by functions depending on the constructor arguments $x_{i1},\ldots,x_{iki}$, as described in Section 4.1.3.

As an example, let `cv` be a variable or constant of type `coord`, as above. Then the goal

```
sum = (case cv of Dim2 a b ⇒ a + b | Dim3 a b c ⇒ a + b + c)
```

is split by the split rule `coord.split` into the goals

```
cv = Dim2 a b ⟹ sum = a + b
cv = Dim3 a b c ⟹ sum = a + b + c
```

Induction Rule

Every definition for a datatype `name` introduces an induction rule (see Section 2.4.5) of the form

```
name.induct:
  ⟦⋀x_1 ... x_k1. ⟦?P x_l1; ... ?P x_lm1⟧ ⟹ ?P (cname_1 x_1 ... x_k1);
   ... ;
   ⋀x_1 ... x_kn. ⟦?P x_ln; ... ?P x_lmn⟧ ⟹ ?P (cname_n x_1 ... x_kn)⟧
  ⟹ ?P ?a
```

where the $x_{l1} \ldots x_{lmi}$ are those $x_1 \ldots x_{ki}$ which have type `name` (i.e., the recursive occurrences of the type name). Like the case rule it is valid because the constructor applications cover all possibilities of constructing a value `?a` of the datatype.

If the datatype `name` is not recursive there are no $x_{l1} \ldots x_{lmi}$ and the assumptions of all inner rules are empty, then the induction rule is simply a specialization of the case rule and is redundant. However, for a recursive datatype `name` induction using rule `name.induct` is the standard way of proving a property to hold for all values.

The rule `name.induct` is associated with datatype `name` for use by the methods `induction` and `induct` (see Section 2.4.5). Therefore the application of the method

```
induction x
```

where *x* is a variable of type *name* splits a goal into *n* subgoals where every subgoal uses a different constructor term in the place of *x*.

As for the case rule and the *cases* method, the names for the named contexts created by the methods *induction* and *induct* are simply the constructor names $cname_i$. Therefore a structured proof using induction for a variable *x* of datatype *name* has the form

```
proof (induction x)
  case (cname₁ x₁ ... x_k1) ... show ?case ⟨proof⟩
next
...
next
  case (cnameₙ x₁ ... x_kn) ... show ?case ⟨proof⟩
qed
```

In the rule *name.induct* all inner assumptions are of the form $?P\ x_{li}$, i.e., they are induction hypotheses and are named "$cname_i$.IH" by the *induction* method, the assumption set "$cname_i$.hyps" is always empty. The *induct* method instead names all inner assumptions by "$cname_i$.hyps".

As an example, the induction rule for the recursive datatype *tree* defined in Section 4.1.1 is

```
tree.induct:
  ⟦⋀x. ?P (Leaf x);
   ⋀x₁ x₂ x₃. ⟦?P x₂; ?P x₃⟧ ⟹ ?P (Tree x₁ x₂ x₃)⟧
  ⟹ ?P ?a
```

If $p :: tree \Rightarrow bool$ is a predicate for values of type *tree* the goal *p x* which states that *p* holds for all values is split by applying the method *(induction x)* into the goals

```
⋀x. p (Leaf x)
⋀x₁ x₂ x₃. ⟦p x₂; p x₃⟧ ⟹ p (Tree x₁ x₂ x₃)
```

A structured proof for goal *p x* has the form

```
proof (induction x)
  case (Leaf x) ... show ?case ⟨proof⟩
next
  case (Tree x₁ x₂ x₃) ... show ?case ⟨proof⟩
qed
```

and in the second case the assumptions $p\ x_2$, $p\ x_3$ are named *Tree.IH*.

4.1.6 Recursive Functions on Algebraic Types

A term is called a "constructor pattern" if it only consists of variables and constructor function applications. Note that this includes terms consisting of a single variable. More generally, a constructor pattern may be a sequence of such terms used as arguments in a function application. A constructor pattern is called "linear" if every variable occurs only once.

HOL provides specific support for recursive definitions (see Section 3.9) of functions *name :: t_1 ⇒ ... ⇒ t_k ⇒ type* where every defining equation *eq_i* has the form

$\bigwedge$ x_{i1} ... x_{ipi}. *name* $term_{i1}$ ... $term_{ik}$ = $term_i$

without assumptions Q_{ij} and the sequence $term_{i1}$ ... $term_{ik}$ on the left side is a linear constructor pattern.

Since the type of a constructor application term is always an algebraic type, an argument type t_j may only be a non-algebraic type if the corresponding $term_{ij}$ is a single variable in all eq_i.

The Proof Method *pat_completeness*

For recursive definitions where all eq_i are without assumptions and using a linear constructor pattern on their left side HOL provides the proof method

pat_completeness

for the proof of equation completeness (see Section 3.9.2), i.e. for solving the first goal created by the recursive definition. The remaining goals for uniqueness can be solved by using the injectivity of the constructor functions which is usually done by the proof method *auto*. Therefore if a recursive definition uses only linear patterns its proof can be specified as

by *pat_completeness auto*

Automatic Uniqueness by Sequential Equations

A constructor pattern *a* is more specific than a constructor pattern *b* if *a* can be constructed from *b* by replacing some variables by constructor applications (and renaming variables). If corresponding function arguments $term_{ij}$ are specified by such patterns *a* and *b* in two defining equations, the spaces of both equations overlap and the defined value must be the same for uniqueness. Alternatively, uniqueness can be guaranteed by replacing the equation with the more general pattern *b* by equations with more specific patterns using the remaining constructors, so that the argument spaces do not overlap anymore.

This can be done automatically for a recursive definition where all eq_i are without assumptions and using a linear constructor pattern on their left side by specifying it in the form

```
function (sequential) name :: "t1 ⇒ ... ⇒ tk ⇒ type"
where eq1 | ... | eqn ⟨proof⟩
```

Here the defining equations must be ordered such that equations with more specific patterns precede those with more general patterns. HOL automatically replaces the latter equations so that the argument spaces of all equations are pairwise disjoint. Note that the resulting equations are used for the completeness and compatibility proofs and also in the rules provided for the recursive definition such as `name.cases` or `name.psimps`.

If the equations do not cover all possile arguments because some constructors are omitted, additional equations are added with patterns using the omitted constructors. These equations use `undefined` (see Section 3.4) as $term_i$ on the right side.

The completeness and compatibility proof must still be specified explicitly, although it always works in the form **by** `pat_completeness auto`.

The *fun* Command

HOL supports the abbreviation

```
fun name :: "t1 ⇒ ... ⇒ tk ⇒ type"
where eq1 | ... | eqn
```

for the recursive definition

```
function (sequential) name :: "t1 ⇒ ... ⇒ tk ⇒ type"
where eq1 | ... | eqn
by pat_completeness auto
termination by lexicographic_order
```

which includes the completeness and compatibility proof and a termination proof. If the termination proof cannot be done by the proof method `lexicographic_order` (see Section 3.9.6) an error is signaled, then the long form must be used to specify another termination proof.

The faculty function definitions in Section 3.9.1 are not of the required form: the definition(s) of `fac` use an assumption in their second equation, the definition of `fac2` uses the argument term `n+1` on the left side which is no constructor pattern because the function `(+)` is not a constructor. However, type `nat` is actually defined in a way equivalent to an algebraic type with constructors `0` and `Suc` (see Section 5.3) where the term `Suc n` is equivalent to `n+1`. Therefore the faculty function can be defined as

```
fun fac3 :: "nat ⇒ nat" where
  "fac3 0 = 1"
| "fac3 (Suc n) = (Suc n) * fac3 n"
```

because `0` and `(Suc n)` are linear constructor patterns.

Primitive Recursion

A linear constructor pattern consisting of a sequence of terms is called "primitive" if exactly one term is a constructor application and all constructor arguments in this term are single variables. Thus a primitive constructor pattern has the general form

$x_1 \ \dots \ x_{(i-1)}$ (cname $x_{i1} \ \dots \ x_{in}$) $x_{(i+1)} \ \dots \ x_k$

where all x_i and x_{ij} are variables. In particular, if a linear constructor pattern consists of a single term it is always primitive.

A recursive function definition

fun *name* :: "t_1 ⇒ ... ⇒ t_k ⇒ *type*"
where eq_1 | ... | eq_n

is called primitive if all defining equations eq_i use a primitive constructor pattern on their left side and all arguments of recursive calls in $term_i$ on the right side are single variables.

Note that since the equations must be ordered so that equations with more specific patterns precede equations with more general patterns all constructor application terms must occur at the same argument position in the patterns. Therefore only one argument type t_i must be an algebraic type, all others may be arbitrary types because the corresponding arguments are denoted by single variables in all patterns.

For primitive recursive function definitions HOL provides the alternative syntax

primrec *name* :: "t_1 ⇒ ... ⇒ t_k ⇒ *type*"
where eq_1 | ... | eq_n

Its difference to the form using the **fun** command is that it only provides the rule set `name.simps` and none of the other rules. The reason is that the `cases` and `induct` rules are mainly equivalent to the case and induction rules provided for the algebraic type of the used constructors (see Section 4.1.5).

Since every such definition can also be written using the **fun** command the use of **primrec** is mainly a documentation that the definition is primitive recursive.

Since the faculty function has only one argument the constructor patterns in the definition of `fac3` above are both primitive. Moreover, since `fac3` is only applied to the plain variable `n` on the right side of the second equation it is a primitive recursive function definition and may be written

```
primrec fac3 :: "nat ⇒ nat" where
  "fac3 0 = 1"
| "fac3 (Suc n) = (Suc n) * fac3 n"
```

4.2 Record Types

Record types resemble algebraic types in that they are roughly equivalent to tuple types, however, they are defined in a completely different way. They do not support recursion, instead they support a simple form of inheritance. They can be used to model "record types" in programming languages and object data in object oriented programming languages.

4.2.1 Record Definitions

A record type is defined in the form

record *rname* = *fname*$_1$ :: "*ftype*$_1$" … *fname*$_n$ :: "*ftype*$_n$"

where `rname` is the name of the new type, the `fname`$_i$ are the pairwise distinct names of the record components (also called "fields") and the `ftype`$_i$ are the corresponding component types. Atleast one component must be specified. The resulting type `rname` is mainly equivalent to the tuple type `ftype`$_1$ × … × `ftype`$_n$ (see Section 3.6), however, every record type definition introduces a new type, even if the component names and types are the same.

The record type defined as above can either be denoted by its `rname` or by its "record type expression" (in inner syntax)

⦇*fname*$_1$ :: *ftype*$_1$, …, *fname*$_n$ :: *ftype*$_n$⦈

Note that a record type expression may only be used after a corresponding record type definition. If there are several record type definitions with the same field names and types the record type expression refers to the syntactically latest previous matching record type definition.

If a field name `fname`$_i$ occurs in several record type definitions it may be referred uniquely as a qualified name by prepending the record type name in the form `rname.fname`$_i$.

An example record definition is

```
record recrd =
  num :: nat
  nums :: "nat set"
  nice :: bool
```

It has an equivalent structure as the datatype `recrd` with the single constructor defined in Section 4.1.2. The record type expression for it is

`⦇num :: nat, nums :: nat set, nice :: bool⦈`

Alternatively the field names may be qualified:

`⦇recrd.num :: nat, recrd.nums :: nat set, recrd.nice :: bool⦈`

Record Type Schemes

To be able to extend a record type by additional fields, a record type definition **record** `rname = fname`$_1$ `:: "ftype`$_1$`" ... fname`$_n$ `:: "ftype`$_n$`"` actually defines a type constructor `rname_scheme` with a single type parameter and an additional component of that type which is called the "more part". Every instantiation of `('a rname_scheme)` is called a record type scheme, the most general one is `('a rname_scheme)` where the more part has an arbitrary type `'a`. For the defined record type the more part has type `unit` (see Section 5.2), i.e., type `rname` is the same as `(unit rname_scheme)`.

Like record types, record type schemes may be denoted by record type expressions. They have the same form, the more part is denoted by the pseudo field name … (three-dot symbol). Therefore the polymorphic type scheme `('a rname_scheme)` can be denoted by the record type expression

`⦇fname`$_1$ `:: ftype`$_1$`, ..., fname`$_n$ `:: ftype`$_n$`, ... :: 'a⦈`

and the record type expression `⦇fname`$_1$ `:: ftype`$_1$`, ..., fname`$_n$ `:: ftype`$_n$`⦈` is simply an abbreviation for `⦇fname`$_1$ `:: ftype`$_1$`, ..., fname`$_n$ `:: ftype`$_n$`, ... :: unit⦈`.

The polymorphic record type scheme for the record type `recrd` defined above is denoted by `'a recrd_scheme` or by the record type expression `⦇num :: nat, nums :: nat set, nice :: bool, ... :: 'a⦈`.

Record type schemes may be used to define functions which are applicable to a record and all its extensions, similar to methods in object oriented programming.

Extending Record Types

A record type is extended by instantiating the more part to a "record fragment type". Like a record type it consists of a sequence of one or more fields, however it cannot be used on its own, it must always be embedded as more part in another record.

A record type is extended by the definition:

record *rname* = *"rtype"*
          + *efname*$_1$ :: *"eftype*$_1$*"* ... *efname*$_m$ :: *"eftype*$_m$*"*

where *rtype* is a previously defined record type. The fields specified by the *efname*$_i$ and *eftype*$_i$ comprise the record fragment which is appended after the fields of *rtype*, even if (some of) the same field names have already been used for *rtype*. Type *rtype* is called the "parent type" of type *rname*. A record type which does not extend another record type (has no parent type) is called a "root record type".

If necessary, the field names *efname*$_i$ must be qualified by *rname*, whereas the field names of the parent type *rtype* must be qualified by the name of *rtype*, even if they occur in an extension of *rtype*.

The record fragment type defined by the extension above may be denoted by the record type expression ⦇*efname*$_1$::*eftype*$_1$, ..., *efname*$_m$::*eftype*$_m$⦈. If ⦇*fname*$_1$::*ftype*$_1$, ..., *fname*$_n$::*ftype*$_n$⦈ is an expression for the parent type, the extended record type may be denoted by the type scheme expression

⦇*fname*$_1$::*ftype*$_1$, ..., *fname*$_n$::*ftype*$_n$,
   ... ::⦇*efname*$_1$::*eftype*$_1$, ..., *efname*$_m$::*eftype*$_m$⦈⦈

or by the type expression

⦇*fname*$_1$::*ftype*$_1$, ..., *fname*$_n$::*ftype*$_n$,
   efname$_1$::*eftype*$_1$, ..., *efname*$_m$::*eftype*$_m$⦈

The example record type defined above can be extended by

record *recrd2 = recrd + full :: bool num :: nat*

Note that the resulting record type has two fields with name *num* and type *nat*, they must always be referred by the qualified field names *recrd.num* and *recrd2.num*. A record type expression for *recrd2* is

⦇*recrd.num :: nat, nums :: nat set, nice :: bool,*
    full :: bool, recrd2.num :: nat⦈

Parameterized Record Types

Like declared types record types may be parameterized (see Section 2.1.2):

record *('name*$_1$*,...,'name*$_n$*) rname = fname*$_1$ :: *"ftype*$_1$*"* ...
                                          fname$_n$ :: *"ftype*$_n$*"*

where the *'name*$_i$ are the type parameters. They may occur in the component types *ftype*$_i$, i.e., the *ftype*$_i$ may be polymorphic (see Section 2.1.3). As usual, the parentheses may be omitted if there is only one type parameter.

For a parameterized record type a record type expression may be specified for every possible instance. As usual, a record type expression with type variables denotes a polymorphic type.

In the record type scheme the parameter for the more part follows all other type parameters.

As an example, a parameterized record type with two type parameters is defined by

record `('a, 'b) recrdp = f1 :: 'a f2:: "'a set" f3 :: 'b`

The most general record type scheme is `('a, 'b, 'c) recrdp_scheme` where `'c` is the parameter for the more part.

After this definition valid record type expressions are ⦇`f1::nat, f2::nat set, f3::bool`⦈ which is equivalent to `(nat, bool) recrdp`, and ⦇`f1::bool, f2::bool set, f3::'a`⦈ which is equivalent to `(bool, 'a) recrdp`, but not ⦇`f1::nat, f2::bool set, f3::bool`⦈, because here the type parameter `'a` has been substituted by the two different types `nat` and `bool`.

4.2.2 Record Constructors

Every record type definition **record** `rname = fname`$_1$ `:: "ftype`$_1$`"` ... `fname`$_n$ `:: "ftype`$_n$`"` defines the record constructor function

`make :: ftype`$_1$ ⇒ ... ⇒ `ftype`$_n$ ⇒ `rname`

which constructs values of the record type from values to be used for all fields. If more than one record type has been defined the name of the constructor function must be qualified by the record type name as `rname.make`.

For an extended record type defined by **record** `rname2 = rname + efname`$_1$ `:: "eftype`$_1$`"` ... `efname`$_m$ `:: "eftype`$_m$`"` the constructor function is

`make :: ftype`$_1$ ⇒ ... ⇒ `ftype`$_n$ ⇒ `eftype`$_1$ ⇒ ... ⇒ `eftype`$_m$ ⇒ `rname2`

It takes values for *all* fields and constructs a full record of type `rname2`.

Every definition for a (root or extended) record type `rname` also defines the constructor function

`extend :: rname ⇒ 'a ⇒ ('a rname_scheme)`

It replaces the more part of a record of type `rname` (which is the unit value) by a value of an arbitrary type `'a`. The result will only be a proper record if `'a` is a record fragment type for a defined extension of `rname`.

Additionally the definition of the extended record type `rname2` defines the constructor function

```
fields :: ftype_{n+1} ⇒ ... ⇒ ftype_m ⇒
       (|fname_{n+1}::ftype_{n+1}, ..., fname_m :: ftype_m|)
```

for the record fragment used as the more part of `rname`. For a root record type `fields` is the same function as `make`.

Constructing Values

Like for a datatype every record constructor function is assumed to be injective, thus their result values differ if atleast one argument value differs. This implies that the set of all values of the constructor function `make` is equivalent to the set of all tuples of values of the field types, which is equivalent to the set of all possible values of the record type `rname`. Thus every value of `rname` may be denoted by a term

`rname.make` $term_1$ ... $term_n$

where each $term_i$ is of type $ftype_i$ and specifies the value for field `i`.

There is an alternative Syntax for applications of record constructors. The record expression

(|$fname_1$ = $term_1$, ..., $fname_n$ = $term_n$|)

denotes the same record value as the constructor application above. If the name $fname_1$ of the first field has been used in more than one record type it must be qualified. The record schema expression

(|$fname_1$ = $term_1$, ..., $fname_n$ = $term_n$, ...= `mterm`|)

denotes a value for the record type scheme where `mterm` denotes the value for the more part.

Values of the record fragment type defined for `rname2` are constructed by the application

`rname2.fields` $term_1$ ... $term_m$

or equivalently by the record fragment expression (|$efname_1$= $term_1$, ..., $efname_m$ = $term_m$|).

Values of the extended record type `rname2` itself can be constructed from a value `r` of type `rname` by `rname.extend r (rname2.fields` $term_1$ ... $term_m$`)` or `rname.extend r` (|`rname2.`$efname_1$= $term_1$, ..., $efname_m$ = $term_m$|).

Values of type `recrd` as defined above are denoted by terms such as `recrd.make 2 {5,7} True` or the equivalent record expression (|`num = 2, nums = {5,7}, nice = True`|) or the record scheme expression (|`num = 2, nums = {5,7}, nice = True, ... = ()`|). Values of the extension `recrd2` as defined above are denoted by terms such as `recrd2.make 2 {5,7} True True 42` or (|`recrd.num = 2, nums = {5,7}, nice = True, full= True, recrd2.num= 42`|) or (|`recrd.num = 2, nums = {5,7}, nice = True, ...=` (|`full= True, recrd2.num= 42`|)|).

4.2.3 Record Destructors

The only record destructors available are selectors which correspond to the field names. Every record type definition **record** *rname = fname*$_1$ *:: "ftype*$_1$*"* … *fname*$_n$ *:: "ftype*$_n$*"* defines the record selector functions

```
fname₁ :: 'a rname_scheme ⇒ ftype₁
 ...
fnameₙ :: 'a rname_scheme ⇒ ftypeₙ
```

Note that instead of *rname* their argument type is the record type scheme *'a rname_scheme*. Thus a record selector function for a field is polymorphic and may also be applied to every extended record to return the field value. However, to make a field name unique, it must be qualified by the name of the record type where it has been introduced.

If *r* is a variable of type *recrd* as defined above, the term *nums r* selects the value of the second field. The same works if *r* has the extended type *recrd2*.

A field selector cannot be applied directly to a record fragment. The fields of the fragment can only be selected if the fragment is embedded in the extended record.

4.2.4 Record Updates

In addition to the constructor and selector functions a record type definition **record** *rname = fname*$_1$ *:: "ftype*$_1$*"* … *fname*$_n$ *:: "ftype*$_n$*"* defines the record update functions

```
fname₁_update :: (ftype₁ ⇒ ftype₁) ⇒ 'a rname_scheme ⇒ 'a rname_scheme
 ...
fnameₙ_update :: (ftypeₙ ⇒ ftypeₙ) ⇒ 'a rname_scheme ⇒ 'a rname_scheme
```

Each record update function *fname*$_i$*_update* takes as argument an update function for values of type *ftype*$_i$ (which maps an old value to a new value) and a record value. It returns the record where the value of field *fname*$_i$ is the result of applying the update function to its old value and the values of all other fields are unchanged.

Like the selector functions the update functions are defined for the polymorphic record type scheme and can thus be also applied to all extended records.

A field may be set to a specific value *term* without regarding the old value by using the term *fname*$_i$*_update (λ_.term) r* where a constant function (see Section 2.1.3) is used as update function for the field value. For record update applications of this form the alternative syntax

```
r⦇ fnameᵢ := term ⦈
```

is available. Further notation for repeated updates is also available: `r(|x := a|)(|y := b|)(|z := c|)` may be written

```
r(|x := a, y := b, z := c|)
```

Note that the former term is equivalent to `z_update (λ_.c) (y_update (λ_.b) (x_update (λ_.a) r))`, so the fields are actually set in the order in which they occur in the alternative notation from left to right.

4.2.5 Record Rules

A record type definition also introduces a number of named facts about the constructors, selectors and update functions. All fact names belong to the namespace of the record definition. The facts are named automatically in the same way for all record types, therefore the fact names must always be qualified by prefixing the record type name.

Several rules are configured for automatic application, e.g., they are added to the simpset for automatic application by the simplifier (see Section 2.3.6). Other rules must be explicitly used by referring them by their name.

Only some basic rules are described here, for more information refer to the Isabelle documentation about records.

Simplifier Rules

The rules for injectivity of the record constructor have the form

```
((|fname₁ = ?x₁, ..., fnameₙ = ?xₙ, ...= ?x|) =
  (|fname₁ = ?y₁, ..., fnameₙ = ?yₙ, ...= ?y|))
= (?x₁ = ?y₁ ∧ ... ∧ ?xₙ = ?yₙ ∧ ?x = ?y)
```

are named `rname.iffs` for the record type `rname`.

Other rules added by a record definition to the simpset process terms where selectors or update functions are applied to constructed record values. They have the form of equations

```
fnameᵢ (|fname₁ = ?x₁, ..., fnameₙ = ?xₙ|) = ?xᵢ
fnameᵢ_update ?f (|fname₁ = ?x₁, ..., fnameₙ = ?xₙ|) =
   (|fname₁ = ?x₁, ..., fnameᵢ = ?f ?xᵢ, ..., fnameₙ = ?xₙ|)
```

The set of all these rules is named `rname.simps` for the record type `rname`.

Additional internal simplifier rules process selectors applied to updated records such as

```
fnameᵢ (fnameᵢ_update ?f ?r) = ?f (fnameᵢ ?r)
fnameᵢ (fnameⱼ_update ?f ?r) = fnameᵢ ?r
```

where `i≠j`.

Constructor Rules

Additional rules provide definitional equations for the constructors of the defined record type, such as

```
rname.make x₁ ... xₙ = ⦇fname₁ = x₁, ..., fnameₙ = xₙ⦈
```

Every rule provides a representation of a constructor application as a record expression. The set of these rules is named `rname.defs` for the record type `rname`. It is not added to the simpset, the rules must be explicitly applied by adding them to the simplifier method when needed (as described in Section 2.3.6).

The simplifier rules are mainly defined for record expressions. To apply them to record terms specified by the constructor functions the constructor rules must be used to convert the terms to record expressions.

Case Rules

Every definition for a record type `rname` introduces case rules (see Section 2.4.3) of the form

```
rname.cases:
  ⟦⋀x₁ ... xₙ. ?y = ⦇fname₁=x₁, ..., fnameₙ=xₙ⦈ ⟹ ?P⟧ ⟹ ?P
rname.cases_scheme:
  ⟦⋀x₁ ... xₙ m. ?y = ⦇fname₁=x₁, ..., fnameₙ=xₙ, ...=m⦈ ⟹ ?P⟧
  ⟹ ?P
```

They are valid because the record expressions cover all possibilities of constructing a value `?y` of the record type or record scheme type, respectively.

Both rules are associated with the record type for use by the `cases` method (see Section 2.4.3), the method automatically selects the most sensible of them. Therefore the application of the method

```
cases "term"
```

where `term` is a record of type `rname` replaces an arbitrary goal by a goal where `term` is set equal to a record constructed from explicit field values `x₁`, ..., `xₙ` and possibly a more part.

The name used for the named context created by the `cases` method is "fields". Therefore a structured proof using case based reasoning for a `term` of a record type `rname` has the form

```
proof (cases "term")
  case (fields x₁ ... xₙ) ... show ?thesis ⟨proof⟩
qed
```

The names x_i of the locally fixed variables can be freely selected, they denote the field values of the record.

The main purpose of applying the case rules is to provide a record expression for a record term which is not specified as such or by a constructor function, such as a variable of a record type. Providing the record expression makes the simplifier rules applicable, therefore a proof for a record often consists of an application of the *cases* method followed by an application of the *simp* method.

Induction Rule

Every definition for a record type *rname* introduces induction rules (see Section 2.4.5) of the form

rname.induct:
  $(\bigwedge x_1 \ldots x_n.\ ?P\ (\!|fname_1{=}x_1, \ldots, fname_n{=}x_n|\!)) \Longrightarrow\ ?P\ ?a$
rname.induct_scheme:
  $(\bigwedge x_1 \ldots x_n\ m.\ ?P\ (\!|fname_1{=}x_1, \ldots, fname_n{=}x_n, \ldots{=}m|\!)) \Longrightarrow\ ?P\ ?a$

Like the case rule it is valid because the record expressions cover all possibilities of constructing a value *?a* of the record type *rname*.

The rules are associated with the record type for use by the methods *induction* and *induct* (see Section 2.4.5), the methods automatically select the most sensible of them. Therefore the application of the method

induction x

where *x* is a variable of type *rname* replaces a goal by a goal which uses a record expression in the place of *x*.

As for the case rule and the *cases* method, the names used for the named contexts created by the methods *induction* and *induct* are “fields”. Therefore a structured proof using induction for a variable *x* of record type *rname* has the form

proof *(induction x)*
  case *(fields* $x_1 \ldots x_n$*)* ... **show** *?case* ⟨*proof*⟩
qed

As an example, the induction rules for the record type *recrd* defined in Section 4.2.1 are

recrd_induct:
  $(\bigwedge x_1\ x_2\ x_3.\ ?P\ (\!|num{=}x_1, nums{=}x_2, nice{=}x_3|\!)) \Longrightarrow\ ?P\ ?a$
recrd_induct_scheme:
  $(\bigwedge x_1\ x_2\ x_3\ m.\ ?P\ (\!|num{=}x_1, nums{=}x_2, nice{=}x_3, \ldots{=}m|\!)) \Longrightarrow\ ?P\ ?a$

By applying the method `(induction x)` the goal `(num x) = y` is replaced by the goal $\bigwedge x_1\ x_2\ x_3$. `(num (|num = `x_1`, nums = `x_2`, nice = `x_3`|)) = y`.
Like the cases methods a transformation of this kind may enable the application of the simplifier methods.

4.3 Subtypes

A subtype specifies the values of a type by a set of values of an existing type. However, since the values of different types are always disjoint, the values in the set are not directly the values of the new type, instead, there is a 1-1 relation between them, they are isomorphic. The values in the set are called “representations”, the values in the new type are called “abstractions”.

4.3.1 Subtype Definitions

A subtype is defined in the form

typedef `name = "term"` ⟨*proof*⟩

where `name` is the name of the new type and `term` is a term for the representing set. The ⟨*proof*⟩ must prove that the representing set is not empty. A subtype definition implies that for every value in the representing set there is a unique value in the defined subtype.

Note that the concept of subtypes actually depends on the specific HOL type `set` for specifying the representing set. See Section 5.4 for how to denote terms for this set. Also note that the set is always of a type `t' set` where `t'` is the common type of all set elements. This implies that the representing set is always a subset of the set of all values of a type `t'` which explains the designation as “subtype”.

A simple example is the type

typedef `three = "{1::nat,2,3}"` **by** `auto`

which has three values. The representations are natural numbers. As usual, the type `nat` must be specified because the constants `1, 2, 3` may also denote values of other types. However, they do not denote the values of the new type `three`, the type definition does not introduce constants for them.

Instead, a subtype definition **typedef** `t = rset` ⟨*proof*⟩ introduces two functions `Abs_t` and `Rep_t`. These are morphisms between `rset` and the new type, `Abs_t` maps from `rset` to type `t`, `Rep_t` is its inverse. Both functions are injective, together they provide the 1-1 mapping between the subtype and the representing set. The function `Abs_t` can be used to denote the values of the subtype. Thus, `Abs_t` plays the role of a constructor for type `t`, whereas `Rep_t` can be thought of being a destructor for `t`.

Actually, if the representing set `rset` is of type `t' set`, the morphism `Abs_t` is a function of type `t' ⇒ t`, since it must be total like all functions in Isabelle. However, `Abs_t` is underspecified as described in Section 2.1.3, no information is given about its result values if applied to values which are not in `rset`.

In the example the morphisms are `Abs_three :: nat ⇒ three` and `Rep_three :: three ⇒ nat`. The values of type `three` may be denoted as `(Abs_three 1)`, `(Abs_three 2)`, and `(Abs_three 3)`. The term `(Abs_three 42)` is a valid term of type `three`, however, no information about its value is available.

Alternative names may be specified for the morphisms in the form

typedef `t = "term"` **morphisms** `rname aname` ⟨*proof*⟩

where `rname` replaces `Rep_t` and `aname` replaces `Abs_t`.

Like declared types subtypes may be parameterized (see Section 2.1.2):

typedef `('name`$_1$`,...,'name`$_n$`) name = "term"` ⟨*proof*⟩

where the `'name`$_i$ are the type parameters. They may occur in the type of the `term`, i.e., the `term` may be polymorphic (see Section 2.1.3).

4.3.2 Type Copies

A type copy is the special case of a subtype definition where the representing set is the universal set (see Section 5.4.1) of another type `t'`:

typedef `t = "UNIV :: t' set"` **by** `auto`

The non-emptiness proof can always be performed by the `auto` method, since the universal set covers all values in type `t'` and types are always non-empty.

The result is a type `t` which is distinct from `t'` but is "isomorphic" to it. The values are in 1-1 relation, although, as usual for distinct types, the value sets are disjoint.

4.3.3 Subtype Rules

A subtype definition only introduces a small number of rules, no rules are added to the simpset.

Basic Morphism Rules

The two morphisms of a subtype definition **typedef** `t = rset` ⟨*proof*⟩ are characterized to be inverses of each other by two rules of the form

```
Abs_t_inverse:
  ?y ∈ rset ⟹ Rep_t (Abs_t ?y) = ?y
Rep_t_inverse:
  Abs_t (Rep_t ?x) = ?x
```

This implies that both morphisms are injective which is stated explicitly by two rules of the form

```
Abs_t_inject:
  ⟦?y₁ ∈ rset; ?y₂ ∈ rset⟧ ⟹ (Abs_t ?y₁ = Abs_t ?y₂) = (?y₁ = ?y₂)
Rep_t_inject:
  (Rep_t ?x₁ = Rep_t ?x₂) = (?x₁ = ?x₂)
```

Since all values of type *t* can be denoted as *Abs_t y* for some *y* in the representing set *rset*, the rule *Abs_t_inject* can be used to prove equality or inequality for values of type *t* based on the equality for values in *rset*.

Case Rules

Every subtype definition **typedef** *t = rset* ⟨*proof*⟩ introduces a case rule (see Section 2.4.3) of the form

```
Abs_t_cases:
  (⋀y. ⟦?x = Abs_t y; y ∈ rset⟧ ⟹ ?P) ⟹ ?P
```

It is valid because the *Abs_t* application covers all possibilities of constructing a value *?x* of the subtype.

The rule *Abs_t_cases* is associated with the new subtype *t* for use by the *cases* method (see Section 2.4.3). Therefore the application of the method

```
cases "term"
```

where *term* is of type *t* applies *Abs_t_cases* to replace the current goal. Since the rule has only one case, it does not split the goal. Applying it to a goal ⋀ x_1 ... x_m. ⟦A_1; ...; A_n⟧ ⟹ *C* as described in Section 2.4.3 results in the single new goal

```
⋀ x₁ ... xₘ y. ⟦A₁; ...; Aₙ; term = Abs_t y; y ∈ rset⟧ ⟹ C
```

where the variable *y* and the two assumptions from the case rule have been added. Together the new goal provides a representation of *term* by applying *Abs_t* to a value *y* from the representing set *rset*. This may allow to use facts about *y* to prove the goal.

The name for the named context created by the *cases* method is simply the morphism name *Abs_t*. Therefore a structured proof using case based reasoning for a *term* of subtype *t* has the form

```
proof (cases "term")
  case (Abs_t y) ... show ?thesis ⟨proof⟩
qed
```

The name *y* of the locally fixed variable can be freely selected, it denotes the morphism argument, i.e., the representation value for *term*.

Every subtype definition **typedef** *t = rset* ⟨*proof*⟩ also introduces an elimination rule (see Section 2.4.4) of the form

```
Rep_t_cases:
  ⟦?y ∈ rset; ⋀x. ?y = Rep_t x ⟹ ?P⟧ ⟹ ?P
```

It is valid because the *Rep_t* application covers all possibilities to determine a representation value *?y* in *rset*.

With the help of this rule it is possible to introduce an abstraction value *x* corresponding to a representation value *?y*, consuming an assumption or input fact that *?y* is in *rset*. For application by the method *cases* the rule is annotated by *[consumes 1]* and the name for the created named context is the morphism name *Rep_t*. As described in Section 2.4.4 a pattern for using the rule in a structured proof is

```
theorem "C" if "y ∈ rset"
  using that
proof (cases rule: Rep_t_cases)
  case (Rep_t x) ... show ?thesis ⟨proof⟩
qed
```

Induction Rules

Every subtype definition **typedef** *t = rset* ⟨*proof*⟩ introduces two induction rules (see Section 2.4.5) of the form

```
Abs_t_induct:
  (⋀y. y ∈ rset ⟹ ?P (Abs_t y)) ⟹ ?P ?a
Rep_t_induct:
  ⟦?a ∈ rset; ⋀x. ?P (Rep_t x)⟧ ⟹ ?P ?a
```

The former rule is a plain induction rule, the latter is an induction rule with elimination where the major premise states that the value *?a* is in *rset*. Both rules only contain a “base case” and no “induction step” with a recursive occurrence of values of the defined type *t*. Like for the case rules they are valid because the morphism applications cover all possibilities of constructing values of *t* or values in *rset*, respectively.

Since the rules only consist of a base case they are mainly equivalent to the case rules. However, when applied by the *induct* method, they not only provide a representation by a morphism for a specified variable, they also substitute every occurrence of the variable by the morphism representation.

The rule *Abs_t_induct* is associated with subtype *t* for use by the methods *induction* and *induct* (see Section 2.4.5). Therefore the application of the method

```
induction x
```

where *x* is a variable of type *t* replaces a goal by a goal where every occurrence of *x* is substituted by the term *Abs_t y* and *y* is a new bound variable with the additional assumption $y \in rset$ named *Abs_t.hyps*. As usual for the induction methods, *x* is substituted in the goal conclusion and also in all goal assumptions.

As for the case rule and the *cases* method, the name for the named context created by the methods *induction* and *induct* is simply the morphism name *Abs_t*. Therefore a structured proof using induction for a variable *x* of subtype *t* has the form

```
proof (induction x)
  case (Abs_t y) ... show ?case ⟨proof⟩
qed
```

As an example, the induction rule for the subtype *three* defined in Section 4.3.1 is

```
Abs_three_induct:
  "⋀y. y ∈ {1, 2, 3} ⟹ ?P (Abs_three y)) ⟹ ?P ?a
```

By applying the method *(induction x)* the goal *x = Abs_three 0* ⟹ *x* ≠ *Abs_three 1* is replaced by the goal ⋀*y*. ⟦*y* ∈ *{1, 2, 3}*; *Abs_three y = Abs_three 0*⟧ ⟹ *Abs_three y* ≠ *Abs_three 1* (which does not help for the proof, but shows the effect of the induction rule).

The rule *Rep_t_induct* is annotated by *[consumes 1]* for application by the methods *induction* and *induct* and the name for the created named context is the morphism name *Rep_t*. As described in Section **??** a pattern for using the rule in a structured proof is

```
theorem "C" if "y ∈ rset"
  using that
proof (induction rule: Rep_t_induct)
  case (Rep_t x) ... show ?case ⟨proof⟩
qed
```

As an example, the induction rule with elimination for the subtype *three* defined in Section 4.3.1 is

```
Rep_three_induct:
  ⟦?a ∈ {1, 2, 3}; ⋀x. ?P (Rep_three x)⟧ ⟹ ?P ?a
```

4.4 Quotient Types

todo

4.5 Lifting and Transfer

todo

Chapter 5

Isabelle HOL Types

This chapter introduces a small basic part of the types available in HOL.

Most of the types are algebraic types (see Section 4.1). Although some of them are defined differently for technical reasons, they are configured afterwards to behave as if they have been defined as algebraic types. Therefore they are described here using the corresponding datatype definition.

If applicable, the functions described for a type include the specific forms of ordering relations and lattice operations (see Section 3.2) and functions for binder syntax (see Section 3.3).

The semantics of the described functions is either given informally for well-known functions or by a description of the form *name :: type* ≡ *lambda-term*. The latter is often not the actual definition used by HOL for the function, it is only used here for documentation purpose.

5.1 Boolean Values

The type of boolean values is specified equivalent to an algebraic type of the form of the enumeration type (see Section 4.1.2)

datatype *bool = True | False*

The type *bool* plays a special role in HOL since it is the type of all terms which are used as propositions and facts (see Section 2.1.6) in Isabelle. Every object logic used in Isabelle must define a type which plays this role.

5.1.1 Values

Values of type *bool* can directly be denoted by the parameterless constructors *True* and *False*.

The lattice constants *top* and *bot* (see Section 3.2.3) are available for type *bool* and denote the values *True* and *False*, respectively.

5.1.2 Destructors

Since both constructors are constant no selectors can be defined. Discriminators are not required since the constants are already boolean values.

A *case* term for type *bool* has the form

```
case term of True ⇒ term₁ | False ⇒ term₂
```

where *term* is a term of type *bool*.

As an alternative syntax HOL provides the usual form

```
if term then term₁ else term₂
```

5.1.3 Functions

The usual logical functions are defined for type *bool*: *conj, disj, implies, iff* of type *bool ⇒ bool ⇒ bool* with operator names *(∧), (∨), (⟶), (⟷)* and the unary negation *Not* of type *bool ⇒ bool* and operator name *(¬)*. Instead of *(⟷)* the default is to use *(=)* as infix operator.

Functions for Orderings and Lattices

The ordering relations (see Section 3.2.2) and the lattice operations (see Section 3.2.3) are defined for type *bool* so that *False < True*. This implies that *(≤)* is equivalent to *(⟶)* and *(⊓), (⊔)* and also *min, max* are equivalent to *(∧), (∨)*.

The lattice operators $(\bigsqcap)$ and $(\bigsqcup)$ on sets are provided for *bool* by definitions $\bigsqcap A \equiv (False \notin A)$ and $\bigsqcup A \equiv (True \in A)$, so they correspond to conjunction and disjunction over sets, respectively. Note that the metalogic quantifier $\bigwedge$ (see Section 2.1.6) does *not* denote a conjunction operation on sets of boolean values. For nonempty finite sets of boolean values the functions *Min* and *Max* are equivalent to $(\bigsqcap)$ and $(\bigsqcup)$.

Functions for Binder Syntax

The quantifiers are defined as "predicates on predicates":

```
All :: ('a ⇒ bool) ⇒ bool ≡ λP. (P = (λx. True))
Ex :: ('a ⇒ bool) ⇒ bool ≡ λP. (¬ All (λx. ¬ P x))
Uniq :: ('a ⇒ bool) ⇒ bool ≡
  λP. (All (λx. (All (λy. P x ⟶ P y ⟶ y = x))))
Ex1 :: ('a ⇒ bool) ⇒ bool ≡
  λP. (Ex (λx. P x ∧ (All (λy. P y ⟶ y = x))))
```

with the alternative binder syntax $\forall$ *x. bterm* for the application *All (λx. bterm)*, $\exists$ *x. bterm* for *Ex (λx. bterm)*, $\exists_{\leq 1}$*x. bterm* for *Uniq (λx. bterm)*, and $\exists$ *!x. bterm* for *Ex1 (λx. bterm)*. The *Uniq* quantifier states that there is atmost one value satisfying the predicate, the *Ex1* quantifier states that there is exactly one such value.

An iterated application for an n-ary predicate $\lambda x_1 \ldots x_n$. *bterm* can be written in the form $\forall\ x_1 \ldots x_n$. *bterm* for all quantifiers. Like for lambda terms (see Section 2.1.3) types may be specified for (some of) the variables as in $\forall\ (x_1 :: type_1) \ldots (x_n :: type_n)$. *bterm*.

5.1.4 Rules

The rules described here are the usual rules for an algebraic type and introduction / elimination / destruction rules for the functions, and some specific rules for negation. HOL provides many additional rules, see the Isabelle documentation for how to use them in proofs.

Complex proofs using these rules can often be done automatically by the proof method *blast* (see Section 2.3.7).

Algebraic Type Rules

Since there are no selectors for *bool* and all constructors are constant the main simplifier rules are the rule sets *bool.distinct* and *bool.case* (see Section 4.1.5):

```
True ≠ False
False ≠ True
(case True of True ⇒ ?t₁ | False ⇒ ?t₂) = ?t₁
(case False of True ⇒ ?t₁ | False ⇒ ?t₂) = ?t₂
```

The case, split, and induction rules are

```
bool.exhaust:
  ⟦?y = True ⟹ ?P; ?y = False ⟹ ?P⟧ ⟹ ?P
bool.split:
  ?P (case ?t of True ⇒ ?t₁ | False ⇒ ?t₂) =
  ((?t = True ⟶ ?P ?t₁) ∧ (?t = False ⟶ ?P ?t₂))
bool.induct:
  ⟦?P True; ?P False⟧ ⟹ ?P ?a
```

Actually, as automatic case rule for type *bool* instead of *bool.exhaust* the slightly different rule

```
case_split:
  ⟦?Q ⟹ ?P; ¬ ?Q ⟹ ?P⟧ ⟹ ?P
```

is used, as described in Section 2.4.3.

For the alternate case term form described in Section 5.1.2 there is also a split rule:

```
if_split:
  ?P (if ?t then ?t₁ else ?t₂) =
  ((?t ⟶ ?P ?t₁) ∧ (¬ ?t ⟶ ?P ?t₂))
```

Other than `bool.split` this rule is automatically applied by the simplifier (see Section 2.3.6) for splitting `if`-terms.

Rules About Boolean Functions

For the functions described in Section 5.1.3 corresponding introduction rules (see Section 2.3.3), destruction rules (see Section 2.3.4), and elimination rules (see Section 2.4.4) are available. They are present in the rule sets `intro`, `dest`, or `elim`, respectively.

The introduction rules are:

```
conjI: ⟦?P; ?Q⟧ ⟹ ?P ∧ ?Q
disjI1: ?P ⟹ ?P ∨ ?Q
disjI2: ?Q ⟹ ?P ∨ ?Q
notI: (?P ⟹ False) ⟹ ¬ ?P
impI: (?P ⟹ ?Q) ⟹ ?P ⟶ ?Q
iffI: ⟦?P ⟹ ?Q; ?Q ⟹ ?P⟧ ⟹ ?P = ?Q
allI: (⋀x. ?P x) ⟹ ∀x. ?P x
exI: ?P ?x ⟹ ∃x. ?P x
```

The destruction rules are:

```
conjunct1: ?P ∧ ?Q ⟹ ?P
conjunct2: ?P ∧ ?Q ⟹ ?Q
mp: ⟦?P ⟶ ?Q; ?P⟧ ⟹ ?Q
spec: ∀x. ?P x ⟹ ?P ?x
iffD1: ⟦?Q = ?P; ?Q⟧ ⟹ ?P
iffD2: ⟦?P = ?Q; ?Q⟧ ⟹ ?P
```

The rule `mp` is the well known logic rule “modus ponens”.

The elimination rules are:

```
conjE: ⟦?P ∧ ?Q; ⟦?P; ?Q⟧ ⟹ ?R⟧ ⟹ ?R
disjE: ⟦?P ∨ ?Q; ?P ⟹ ?R; ?Q ⟹ ?R⟧ ⟹ ?R
notE: ⟦¬ ?P; ?P⟧ ⟹ ?R
impE: ⟦?P ⟶ ?Q; ?P; ?Q ⟹ ?R⟧ ⟹ ?R
iffE: ⟦?P = ?Q; ⟦?P ⟶ ?Q; ?Q ⟶ ?P⟧ ⟹ ?R⟧ ⟹ ?R
allE: ⟦∀x. ?P x; ?P ?x ⟹ ?R⟧ ⟹ ?R
exE: ⟦∃x. ?P x; ⋀x. ?P x ⟹ ?Q⟧ ⟹ ?Q
```

Additionally, the following four rules can be used for “proofs by contradiction”:

contrapos_pn: $[\![?Q;\ ?P \Longrightarrow \neg\ ?Q]\!] \Longrightarrow \neg\ ?P$
contrapos_pp: $[\![?Q;\ \neg\ ?P \Longrightarrow \neg\ ?Q]\!] \Longrightarrow ?P$
contrapos_nn: $[\![\neg\ ?Q;\ ?P \Longrightarrow ?Q]\!] \Longrightarrow \neg\ ?P$
contrapos_np: $[\![\neg\ ?Q;\ \neg\ ?P \Longrightarrow ?Q]\!] \Longrightarrow ?P$

Equivalence of Derivation Rules and Boolean Terms

Using these rules every derivation rule (see Section 2.1.6)

$\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow C$

can be converted to the boolean term

$\forall\ x_1 \ldots x_m.\ (A_1' \wedge \ldots \wedge A_n') \longrightarrow C$

(where each A_i' is converted in the same way if it is again a derivation rule), and vice versa.

In principle every theorem may be specified in either of both forms. However, its application by proof methods in other proofs is usually only possible if it is specified in derivation rule form. Therefore it is usually preferable to specify theorems in this form, where the conclusion C does not use $\forall$ or $\longrightarrow$ as its outermost operator.

5.2 The Unit Type

The unit type has only one value. It is specified equivalent to an algebraic type of the form of the enumeration type (see Section 4.1.2)

datatype *unit* = *Unity*

A typical use of the unit type is for instantiating a parameter of a parameterized type when you actually don’t care about it. For example, if you have to use binary predicates of type *'a ⇒ 'b ⇒ bool* but do not care about the second argument you can use the type *'a ⇒ unit ⇒ bool*. Syntactically its values are still binary predicates, however, since there is only one possible value for the second argument, it cannot affect the result, so semantically they are unary predicates. Another use of the unit type is for the more part in unextended record types (see Section 4.2.1).

So the unit type plays the role of an “empty type” (which does not exist for formal reasons).

5.2.1 Values

Values of type `unit` can directly be denoted by the parameterless constructor *Unity*. The usual way of denoting it is by its alternative form `()`, which has been chosen because the unit type is also used to represent empty tuples (see Section 5.6).

5.2.2 Destructors

A *case* term for type `unit` has the form

$case\ term\ of\ ()\ \Rightarrow\ term_1$

where *term* is a term of type `unit`. It is equivalent to $term_1$.

5.2.3 Rules

The usual rules for an algebraic type are also defined for `unit`. They are of no much use, but here they are shown for the interested reader:

```
unit.exhaust: (?y = () ⟹ ?P) ⟹ ?P
unit.split: ?P (case ?t of () ⇒ ?t₁) = (?t = () ⟶ ?P ?t₁)
unit.induct: ?P () ⟹ ?P ?a
```

The split rule is required if during a proof a *case* term for type `unit` (see Section 5.2.2) occurs for some reason. Although it is trivially equivalent to its subterm $term_1$, the simplifier will not use that equivalence since it never splits *case* terms automatically. It must be configured as *simp split: unit.split* to do so.

Since there are no functions for `unit` there are no introduction / destruction / elimination rules.

5.3 Natural Numbers

The type of natural numbers is specified equivalent to a recursive algebraic type (see Section 4.1) of the form

datatype *nat = 0 | Suc nat*

It is not really defined in this way, because *0* is syntactically not a constructor, but it can mainly be used in the same way.

The type `nat` denotes the mathematical concept of natural numbers, it has infinitely many values, there is no upper limit.

5.3.1 Values

Values of type `nat` can be denoted in the usual way using constructor expressions such as `Suc 0, Suc (Suc 0), ...`.

Alternatively they can be denoted by decimal number literals such as `0, 1, 2, ...` of arbitrary size.

However, the decimal number literals are overloaded and may also denote values of other numerical types, such as type `int` for the integer numbers. Therefore the type of an isolated decimal number literal is not determined, which may cause unexpected effects. To denote a value of type `nat` its type may be explicitly specified as described in Section 2.1.3, such as `1::nat`.

The lattice constant `bot` (see Section 3.2.3) is available for type `nat` and denotes the value `0`. The constant `top` is not available for type `nat`.

5.3.2 Destructors

Since `Suc` plays the role of a constructor, corresponding destructors can be defined. The selector function which inverts `Suc` is defined as `nat.pred` where `nat.pred x` is equivalent to `x - 1` and `nat.pred 0 = 0`. This selector is not intended to be used directly, use the subtraction function described below instead. There are no discriminators, instead the equality terms `x = 0` and `x ≠ 0` can be used.

A `case` term for type `nat` has the form

```
case term of 0 ⇒ term₁ | Suc n ⇒ term₂
```

where `term` is a term of type `nat`.

5.3.3 Functions

The usual basic arithmetic functions are defined for type `nat`: `plus, minus, times, divide, modulo` of type `nat ⇒ nat ⇒ nat` with operator names `(+), (-), (*), (div), (mod)` and alternate operator name `(/)` for `(div)`. Subtraction is truncated at `0`, e.g. `4 - 7` evaluates to `0`. Also defined is the binary "divides" relation `dvd_class.dvd` with operator name `(dvd)`.

Like decimal number literals all these functions are overloaded and not restricted to natural numbers. As a consequence, a proposition such as

```
4 - 7 = 0
```

is not valid and cannot be proved. To become a provable fact it must be specified in a form like

```
(4::nat) - 7 = 0
```

which can be proved by method *simp*. Note that it is sufficient to specify the type for a single literal, because then the type of the function *(-)* is derived to be *nat* ⇒ *nat* ⇒ *nat* (there are no "mixed-typed" arithmetic functions) from which the type of the second literal is derived and similar for the equality.

For type *nat* HOL defines the *size* function (see Section 3.8.4) to be the identity function.

Functions for Orderings and Lattices

The ordering relation *(<)* (see Section 3.2.2) is defined to be the usual ordering on natural numbers. This implies that also *min* and *max* have their usual meaning. The lattice operations (see Section 3.2.3) are overloaded for type *nat* so that *(⊓), (⊔)* are equivalent to *min, max*.

The functions *Min* and *Max* on sets are also defined as expected on finite nonempty sets of natural numbers, otherwise their result is underspecified. The lattice operators *(⨅)* and *(⨆)* are equivalent to *Min* and *Max*, additionally *⨆{}* is specified to be *0* and *(⨅)* is also specified to return the minimal value for infinite sets of natural numbers.

5.3.4 Rules

HOL provides a large number of rules applicable for proofs about values of type *nat*. Here we only show rules for an algebraic type and introduction / elimination / destruction rules for the functions, like for other types. They are usually not sufficient for proofs about natural numbers and should only give an impression about the type *nat* in comparison with other types.

Proofs for linear arithmetic properties of *nat* values using these and other rules can often be done automatically by the proof methods *linarith* or *arith* (see Section 2.3.7).

Algebraic Type Rules

The simplifier rules for the constructors of type *nat* are the rule sets *nat.inject*, *nat.distinct* and *nat.case* (see Section 4.1.5):

```
(Suc ?x = Suc ?y) = (?x = ?y)
0 ≠ Suc ?x
Suc ?x ≠ 0
(case 0 of 0 ⇒ ?t₁ | Suc x ⇒ ?t₂ x) = ?t₁
(case Suc ?x of 0 ⇒ ?t₁ | Suc x ⇒ ?t₂ x) = ?t₂ ?x
```

Note the difference between the unknown *?x* and the locally bound *x* in the last rule.

The case, split, and induction rules are

```
nat.exhaust:
  ⟦?y = 0 ⟹ ?P; ⋀x. ?y = Suc x ⟹ ?P⟧ ⟹ ?P
nat.split:
  ?P (case ?t of 0 ⇒ ?t1 | Suc x ⇒ ?t2 x) =
  ((?t = 0 ⟶ ?P ?t1) ∧ (∀x. ?t = Suc x ⟶ ?P (?t2 x)))
nat.induct:
  ⟦?P 0; ⋀i. ?P i ⟹ ?P (Suc i)⟧ ⟹ ?P ?n
```

For the case and induction rule the cases are named `0` and `Suc`.

Other Induction Rules

The rule `nat.induct` is the induction rule associated with type `nat`. Additionally, there are several other induction rules for values of type `nat` which may be used for other types of induction by specifying them explicitly to the `induction` or `induct` method:

```
nat_less_induct:
  ⟦⋀i. ∀ j<i. ?P j ⟹ ?P i⟧ ⟹ ?P ?n
infinite_descent0:
  ⟦?P 0; ⋀i. ⟦0 < i; ¬ ?P i⟧ ⟹ ∃ j<i. ¬ ?P j⟧ ⟹ ?P ?n
diff_induct:
  ⟦⋀i. ?P i 0; ⋀j. ?P 0 (Suc j); ⋀i j. ?P i j ⟹ ?P (Suc i) (Suc j)⟧
  ⟹ ?P ?m ?n
less_Suc_induct [consumes 1]:
  ⟦?m < ?n; ⋀i. ?P i (Suc i);
   ⋀i j k. ⟦i < j; j < k; ?P i j; ?P j k⟧ ⟹ ?P i k⟧
  ⟹ ?P ?m ?n
nat_induct_at_least [consumes 1]:
  ⟦?m ≤ ?n; ?P ?m; ⋀i. ⟦?m ≤ i; ?P i⟧ ⟹ ?P (Suc i)⟧ ⟹ ?P ?n
nat_induct_non_zero [consumes 1]:
  ⟦0 < ?n; ?P 1; ⋀i. ⟦0 < i; ?P i⟧ ⟹ ?P (Suc i)⟧ ⟹ ?P ?n
inc_induct [consumes 1]:
  ⟦?n ≤ ?m; ?P ?m; ⋀i. ⟦?n ≤ i; i < ?m; ?P (Suc i)⟧ ⟹ ?P i⟧
  ⟹ ?P ?n
strict_inc_induct [consumes 1]:
  ⟦?n < ?m; ⋀i. ?m = Suc i ⟹ ?P i;
   ⋀i. ⟦i < ?m; ?P (Suc i)⟧ ⟹ ?P i⟧
  ⟹ ?P ?n
dec_induct [consumes 1]:
  ⟦?m ≤ ?n; ?P ?m; ⋀i. ⟦?m ≤ i; i < ?n; ?P i⟧ ⟹ ?P (Suc i)⟧
  ⟹ ?P ?n
```

Note that the last six rules combine induction with elimination, as described in Section 2.4.5, and are therefore attributed by `[consumes 1]`.

Rules About Functions

The constructor function `Suc` has result type `nat` and therefore cannot occur as outermost operator in a proposition. Therefore introduction, destruction, and elimination rules for `Suc` must embed the application term in a proposition with the help of a predicate. The ordering relations are used for this purpose, thus only ordering properties can be proved using these rules.

Introduction rules (see Section 2.3.3) for `Suc` are

```
le_SucI: ?m ≤ ?n ⟹ ?m ≤ Suc ?n
less_SucI: ?m < ?n ⟹ ?m < Suc ?n
Suc_leI: ?m < ?n ⟹ Suc ?m ≤ ?n
Suc_lessI: ⟦?m < ?n; Suc ?m ≠ ?n⟧ ⟹ Suc ?m < ?n
```

Destruction rules (see Section 2.3.4) for `Suc` are

```
Suc_leD: Suc ?m ≤ ?n ⟹ ?m ≤ ?n
Suc_lessD: Suc ?m < ?n ⟹ ?m < ?n
Suc_less_SucD: Suc ?m < Suc ?n ⟹ ?m < ?n
Suc_le_lessD: Suc ?m ≤ ?n ⟹ ?m < ?n
```

An elimination rule (see Section 2.4.4) for `Suc` is

```
less_SucE:
⟦?m < Suc ?n; ?m < ?n ⟹ ?P; ?m = ?n ⟹ ?P⟧ ⟹ ?P
```

For the arithmetic functions described in Section 5.3.3 most properties are specified by equations such as

```
add_Suc_right: ?m + Suc ?n = Suc (?m + ?n)
mult_Suc_right: ?m * Suc ?n = ?m + ?m * ?n
diff_Suc_1: Suc ?n - 1 = ?n
min_Suc_Suc: min (Suc ?m) (Suc ?n) = Suc (min ?m ?n)
max_Suc_Suc: min (Suc ?m) (Suc ?n) = Suc (min ?m ?n)
dvd_diff_nat: ⟦?k dvd ?m; ?k dvd ?n⟧ ⟹ ?k dvd ?m - ?n
```

Many of these equations are members of the simpset, therefore many simple arithmetic properties can be proved by the simplifier. More complex properties may need the methods `linarith` or `arith`, or can be proved by induction or a combination thereof.

5.4 Sets

You may think of the type constructor `set` as being specified equivalent to the parameterized algebraic type

datatype `'a set = Collect (contains: 'a ⇒ bool)`

Thus a set is simply a wrapper for a unary predicate of type `'a ⇒ bool` (see Section 3.1.1).

The selector `contains` is not intended for general use and can only be specified in qualified form as `Predicate_Compile.contains`.

5.4.1 Values

Basically, values of type `'a set` are denoted by constructor application terms of the form `Collect (λx. bterm)` where `(λx. bterm)` is a unary predicate (see Section 3.1.1).

For such a term HOL provides the alternative "set comprehension" syntax (which is a special form of binder syntax)

```
{x. bterm}
```

and the abbreviations `{}` for the empty set `{x. False}` and `UNIV` for the universal set `{x. True}`. Both abbreviations are available for arbitrary types `'a set`. The universal set is the set of all values of the type `'a`. Examples are `UNIV :: bool set` which is the set `{True, False}` and `UNIV :: nat set` which is the set of all natural numbers.

The lattice constants `top` and `bot` (see Section 3.2.3) are available for sets and denote the universal set `UNIV` and the empty set `{}`, respectively.

For a set comprehension of the form

$\{x.\ \exists\ x_1\ \ldots\ x_n.\ x = term \land bterm\}$

HOL provides the alternative syntax

$\{term\ |\ x_1\ \ldots\ x_n.\ bterm\}$

Note that the bindings of the variables $x_1\ \ldots\ x_n$ follow the `|` although the scope of the bindings includes the `term` before the `|`.

An example for this syntax is the set

```
{2*x | x::nat. x < 5}
```

which contains the numbers `0, 2, 4, 6, 8`. Note that types may be specified in the bindings in the same way as for the ∃ quantifier.

5.4.2 Destructors

For use instead of the selector

```
Predicate_Compile.contains :: 'a set ⇒ 'a ⇒ bool
```

HOL provides the function with reversed arguments

```
Set.member :: 'a ⇒ 'a set ⇒ bool
```

with alternate operator name *(∈)*. It combines unwrapping and applying the predicate and corresponds to the usual member test predicate for sets.

Since there is only one constructor, discriminators and case terms are not available for sets.

5.4.3 Functions

In addition to the basic function `Set.member` and its negation `Set.not_member` with operator name *(∉)* HOL provides the other usual functions on sets: the relations `subset`, `subset_eq`, `supset`, `supset_eq` with operator names *(⊂)*, *(⊆)*, *(⊃)*, *(⊇)* and the operations `inter`, `union`, `minus` with operator names *(∩)*, *(∪)*, *(-)*. The function `minus` is set difference, there is also the unary set complement (relative to the universal set `UNIV`, see Section 5.4.1) `uminus` which also has the operator name *(-)* for prefix application. The functions `minus` and `uminus` are overloaded for other types as well, therefore Isabelle will not automatically derive that their arguments are sets.

Intersection and union of a family of sets is supported by the functions

```
Inter :: 'a set set ⇒ 'a set
Union :: 'a set set ⇒ 'a set
```

with operator names *(⋂)* and *(⋃)* for prefix notation.

HOL also provides the function

```
insert :: 'a ⇒ 'a set ⇒ 'a set ≡ λa B. {x. x = a ∨ x ∈ B}
```

which inserts a value into a set, together with the set enumeration notation

$\{x_1,\ \ldots,\ x_n\}$

as abbreviation for `insert` x_1 *(... (*`insert` x_n *{}) ...)*.

Moreover HOL provides the functions

```
Pow :: 'a set ⇒ 'a set set ≡ λA. {B. B⊆A}
image :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ≡ λf A. {y. ∃x∈A. y = f x}
range :: ('a ⇒ 'b) ⇒ 'b set ≡ λf. image f UNIV
vimage :: 'a ⇒ 'b) ⇒ 'b set ⇒ 'a set ≡ λf A. {x. f x ∈ A}
is_singleton :: 'a set ⇒ bool ≡ λA. (∃x. A = {x})
the_elem :: 'a set ⇒ 'a ≡ λA. (THE x. A = {x})
pairwise :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool ≡
  λf A. (∀x∈A. ∀y∈A. x ≠ y ⟶ f x y)
disjnt :: 'a set ⇒ 'a set ⇒ bool ≡ λA B. A ∩ B = {}
```

with operator name *(`)* for `image` and *(-`)* for `vimage`. `Pow` is the powerset operator, `image` and `vimage` are the image / reverse image of a function. `range` is the set of all result values of a function (note that the set of all argument values of a function is always *UNIV* because functions are total in Isabelle). As usual, the result of *the_elem A* is underspecified (see Section 3.3.2) if *A* is not known to be a singleton set. The relation application *pairwise f A* is satisfied if *f* is satisfied for all pairs of different elements of *A*. The relation *disjnt* tests two sets for being disjoint.

As a convention, variables for sets are usually denoted by uppercase letters.

Functions for Orderings and Lattices

The ordering relations (see Section 3.2.2) are defined for sets so that *(<)* is equivalent to the subset relation $(\subset)$ and analogously for $(\leq)$, $(>)$, $(\geq)$. The lattice operations $(\sqcap)$, $(\sqcup)$, $(\bigsqcap)$, and $(\bigsqcup)$ (see Section 3.2.3) are overloaded to be equivalent to $(\cap)$, $(\cup)$, $(\bigcap)$, and $(\bigcup)$.

Since $(\subset)$ is not a total ordering the functions *min* and *max* are *not* equivalent to $(\cap)$ and $(\cup)$ and the functions *Min* and *Max* are not available for sets of sets.

Functions for Binder Syntax

HOL provides many functions which support binder syntax where a single bound variable is restricted ("bounded") by a member or subset relation to some set.

HOL provides functions similar to *All* and *Ex* (see Section 5.1.3) to support the syntax of bounded quantifiers

```
∀x∈sterm. bterm ≡ ∀x. x ∈ sterm ⟶ bterm
∃x∈sterm. bterm ≡ ∃x. x ∈ sterm ∧ bterm
∃!x∈sterm. bterm ≡ ∃!x. x ∈ sterm ∧ bterm
∀x⊂sterm. bterm ≡ ∀x. x ⊂ sterm ⟶ bterm
∃x⊂sterm. bterm ≡ ∃x. x ⊂ sterm ∧ bterm
∀x⊆sterm. bterm ≡ ∀x. x ⊆ sterm ⟶ bterm
∃x⊆sterm. bterm ≡ ∃x. x ⊆ sterm ∧ bterm
∃!x⊆sterm. bterm ≡ ∃!x. x ⊆ sterm ∧ bterm
```

and the bounded descriptors (see Section 3.3.3)

```
LEAST x∈sterm. bterm ≡ LEAST x. x ∈ sterm ∧ bterm
GREATEST x∈sterm. bterm ≡ GREATEST x. x ∈ sterm ∧ bterm
```

Unlike for the plain quantifiers only one bounded variable may be specified for these forms. If there are more, the quantifiers must be nested as in $\forall x{\in}sterm_1.\ \forall y{\in}sterm_2.\ bterm$.

As set comprehension syntax for the special case of a predicate which includes a member test HOL provides the syntax

```
{x∈sterm. bterm}
```

for a term of the form `{x. x∈sterm ∧ bterm}`.

For the operations `(⋂)`, `(⋃)`, `(⨅)`, `(⨆)`, `Min`, `Max` on sets (see Section 3.2) HOL provides the alternative syntax of the form

```
⋂x∈term₁. term₂
```

where both terms must have a `set` type and `x` may occur free in `term₂`. This form is equivalent to `⋂{term₂ | x. x∈term₁}` which is the intersection over all sets returned by `term₂` when `x` adopts all values in the set `term₁`. For `Min` the syntax is

```
MIN x∈term₁. term₂
```

and analogously for `Max`.

For all these operators HOL also provides the abbreviated syntax of the form

```
⋂x. term₂
```

for `⋂x∈UNIV. term₂` and the further abbreviation

```
⋂x₁ ... xₙ. term₂
```

for `⋂x₁. ... ⋂xₙ. term₂` (which is not available for the form with ∈).

Note that `MIN x::nat. x < 5` is *not* the minimum of the numbers which are less than `5`, instead it is the minimum of the boolean values which occur as the result of `x < 5` if `x` adopts all possible values of type `nat`, i.e. it is equal to the value `False`. The minimum of the numbers which are less than `5` is denoted by `LEAST x::nat. x < 5` which is equal to the value `0`.

Functions for Finite Sets

HOL defines the predicate `finite` by the inductive definition (see Section 3.7)

```
inductive finite :: "'a set ⇒ bool" where
  "finite {}"
| "finite A ⟹ finite (insert a A)"
```

So a set is finite if it can be constructed from the empty set by a finite sequence of inserting single elements.

For iterating through the elements of a finite set HOL introduces the function

```
Finite_Set.fold :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b
```

where `fold f z {`x_1`, ..., `x_n`} = f `x_1` (... (f `x_n` z)...)` if the resulting value is independent of the order of the x_i, i.e., the function `f` must be "left-commutative" on the values in the set. If it is not, its result is underspecified, if the set is not finite the result is the starting value `z`.

The function `Finite_Set.fold` is not intended for direct use, it is used by HOL to provide other functions. The most basic is

```
card :: 'a set ⇒ nat ≡ λA. Finite_Set.fold (λ_ n. Suc n) 0 A
```

for the cardinality of sets. For finite sets it is the number of elements, for infinite sets, due to the way `fold` is defined, it is always `0`.

5.4.4 Rules

Algebraic Type Rules

Since HOL does not define the type `'a set` as an algebraic type it does not provide the standard rules and rule sets. However, the relevant properties are available in slightly different form.

Injectivity of the constructor is not specified as a simplifier equation, instead there is the rule

```
Collect_inj: Collect ?P = Collect ?Q ⟹ ?P = ?Q
```

Note that the other direction is always satisfied for arbitrary functions (see Section 5.7.3). It is provided in the form where the predicates are applied to arguments:

```
Collect_eqI: (⋀x. ?P x = ?Q x) ⟹ Collect ?P = Collect ?Q
```

where its name reflects that it has the form of an introduction rule (see Section 2.3.3) for the equality relation `(=)` (see Section 3.2.1) on type `'a set`.

Equivalent to a "defining equation" for the selector there is the rule

```
mem_Collect_eq: Set.member ?a (Collect ?P) = ?P ?a
```

which is

```
(?a ∈ {x. ?P x}) = ?P ?a
```

in alternative syntax. This rule is also provided in the form of two separate rules for both dirctions:

```
CollectI: ?P ?a ⟹ ?a ∈ {x. ?P x}
CollectD: ?a ∈ {x. ?P x} ⟹ ?P ?a
```

having the form of an introduction and a destruction rule (see Section 2.3.4).

Due to the simple wrapper structure of type `'a set` no other algebraic type rules apply.

Rules About Functions

Using the rules described above and the definitions of the functions on sets described in Section 5.4.3 it is possible to convert every proposition about sets to an equivalent proposition about predicates and boolean operations (see Section 5.1.3) and prove it in this form. Most automatic proof methods described in Section 2.3.7 use this conversion and can thus prove many goals which involve sets and set operations, in particular the method *blast*.

Additionally HOL provides rules which directly work for functions on sets such as the introduction rules (see Section 2.3.3)

```
subsetI: (⋀x. x ∈ ?A ⟹ x ∈ ?B) ⟹ ?A ⊆ ?B
IntI: ⟦?c ∈ ?A; ?c ∈ ?B⟧ ⟹ ?c ∈ ?A ∩ ?B
UnI1: ?c ∈ ?A ⟹ ?c ∈ ?A ∪ ?B
PowI: ?A ⊆ ?B ⟹ ?A ∈ Pow ?B
```

the destruction (see Section 2.3.4) and elimination (see Section 2.4.4) rules

```
subsetD: ⟦?A ⊆ ?B; ?c ∈ ?A⟧ ⟹ ?c ∈ ?B
IntD1: ?c ∈ ?A ∩ ?B ⟹ ?c ∈ ?A
UnE: ⟦?c ∈ ?A ∪ ?B; ?c ∈ ?A ⟹ ?P; ?c ∈ ?B ⟹ ?P⟧ ⟹ ?P
PowD: ?A ∈ Pow ?B ⟹ ?A ⊆ ?B
```

and the simplifier rules (see Section 2.3.6)

```
subset_eq: (?A ⊆ ?B) = (∀x∈?A. x ∈ ?B)
Int_def: ?A ∩ ?B = {x. x ∈ ?A ∧ x ∈ ?B}
Un_def: ?A ∪ ?B = {x. x ∈ ?A ∨ x ∈ ?B}
Pow_def: Pow ?A = {B. B ⊆ ?A}
```

where, despite their names, *(∩)* and *(∪)* are not really defined in this way.

There are also many rules about *finite* and *card* such as

```
finite_subset: ⟦?A ⊆ ?B; finite ?B⟧ ⟹ finite ?A
finite_Un: finite (?F ∪ ?G) = (finite ?F ∧ finite ?G)
finite_Int: finite ?F ∨ finite ?G ⟹ finite (?F ∩ ?G)
card_mono: ⟦finite ?B; ?A ⊆ ?B⟧ ⟹ card ?A ≤ card ?B
card_Diff_subset_Int:
  finite (?A ∩ ?B) ⟹ card (?A - ?B) = card ?A - card (?A ∩ ?B)
card_Un_Int:
  ⟦finite ?A; finite ?B⟧
  ⟹ card ?A + card ?B = card (?A ∪ ?B) + card (?A ∩ ?B)
```

Many rules about *finite* are known by the automatic proof methods (see Section 2.3.7). Rules about *card* must often be specified explicitly.

5.5 Optional Values

A function argument is optional if it can be omitted. In Isabelle, however, every function has a fixed number of arguments, it is not possible to omit one or more of them. Instead, the idea is to have a special value with the meaning of “no value”. The presence of such a value is no more a property of the function, it is a property of the function argument’s *type*.

Normally, types do not include a “no value” value, it must be introduced separately and it must be different from all existing values. For a type `t` this can be done by defining a new type which has one more value. However, since the values of the new type are always different from those of `t` (see Section 2.1.2), the new type has to include the values of `t` in a “wrapped” form.

All this is done by the algebraic type

```
datatype 'a option =
  None
| Some (the: 'a)
```

It is polymorphic with one type parameter `'a` (see Section 2.1.2), for every type `t` it provides the new type `t option`.

In this way the type `nat option` can be used to add a “no value” to the natural numbers.

The type constructor `option` can also be used to emulate partial functions (which do not exist in Isabelle) by functions with a result type `t option`.

5.5.1 Values

Type `option` provides two constructors. The parameterless constructor `None` denotes the “no value” value. The unary constructor `Some` provides the value `Some v` for every value `v` of the type parameter `'a`. In this way the type `t option` includes all values `v` of `t` wrapped as `Some v`. Since constructors of algebraic types are injective (see Section 4.1.2) different values stay different when wrapped. Since different constructors denote different values the value `None` is different from all wrapped values. Note also, that for different types `t1` and `t2` the `None` values in `t1 option` and `t2 option` are different, because they belong to different types.

The type `nat option` provides the value `None` and wrapped natural numbers of the form `Some 5`.

5.5.2 Destructors

The selector `the` is used to “unwrap” the wrapped values: `the (Some v) = v`. As described in Section 4.1.3 the selector can also be applied to `None`, but the result is underspecified. The term `the None` denotes a unique value of the parameter type `'a`, but no information is available about that value.

Although not introduced by the type definition, the function `Option.is_none` plays the role of a discriminator and tests values for being `None`. Alternatively the equality terms `x = None` and `x ≠ None` can be used.

A `case` term for type `t option` has the form

```
case term of None ⇒ term₁ | Some v ⇒ term₂
```

where `term` is a term of type `t option` and `v` is a variable of type `t` which may occur in `term₂`. Such `case` terms can be used to process a `term` for an optional value. It is tested whether a (wrapped) value is present, if not `term₁` specifies a default, otherwise `term₂` specifies a use of the unwrapped value `v`.

The same effect can be achieved by the term

```
if term = None then term₁ else term₂'
```

where `term₂'` uses `(the term)` instead of `v`. However, proofs may differ depending on which of both forms is used.

5.5.3 Functions

The function

```
Option.these :: 'a option set ⇒ 'a set ≡
  λA. image the {x∈A. x≠None}
```

extends the selector `the` to sets. It returns the set of unwrapped values from a set of optional values, ignoring `None` if present in the set.

No orderings or lattice functions (see Section 3.2) are specified for values of type `'a option`.

BNF Functions

The type constructor `option` is a bounded natural functor as described in Section 4.1.4. Values of type `'a option` can be viewed as containers of a single value of type `'a`.

The corresponding BNF functions are generated as:

```
set_option :: 'p option ⇒ 'p set ≡
  λx. case x of None ⇒ {} | Some x' ⇒ {x'}
map_option :: ('p ⇒ 'q) ⇒ 'p option ⇒ 'q option ≡
  λf x. case x of None ⇒ None | Some x' ⇒ Some (f x')
pred_option :: ('p ⇒ bool) ⇒ 'p option ⇒ bool ≡
  λp x. case x of None ⇒ True | Some x' ⇒ p x'
rel_option :: ('p ⇒ 'q ⇒ bool) ⇒ 'p option ⇒ 'q option ⇒ bool ≡
  λr x y. case x of None ⇒ y = None | Some x' ⇒
            (case y of None ⇒ False | Some y' ⇒ r x' y')
```

Additionally there is the function

```
combine_options ::
  ('p ⇒ 'p ⇒ 'p) ⇒ 'p option ⇒ 'p option ⇒ 'p option ≡
  λf x y. case x of None ⇒ y | Some x' ⇒
            (case y of None ⇒ x | Some y' ⇒ Some (f x' y'))
```

which extends `map_option` to binary functions. It applies its first argument (a binary function) to two wrapped values without unwrapping them. If one argument is `None` the result is the other argument.

5.5.4 Rules

Algebraic Type Rules

The simplifier rules for the constructors of type `'a option` are the rule sets `option.inject`, `option.distinct` and `option.case` (see Section 4.1.5):

```
(Some ?x = Some ?y) = (?x = ?y)
None ≠ Some ?x
Some ?x ≠ None
(case None of None ⇒ ?t₁ | Some x ⇒ ?t₂ x) = ?t₁
(case Some ?x of None ⇒ ?t₁ | Some x ⇒ ?t₂ x) = ?t₂ ?x
```

The case, split, and induction rules are

```
option.exhaust:
  ⟦?y = None ⟹ ?P; ⋀x. ?y = Some x ⟹ ?P⟧ ⟹ ?P
option.split:
  ?P (case ?t of None ⇒ ?t₁ | Some x ⇒ ?t₂ x) =
  ((?t = None ⟶ ?P ?t₁) ∧ (∀x. ?t = Some x ⟶ ?P (?t₂ x)))
option.induct:
  ⟦?P None; ⋀x. ?P (Some x)⟧ ⟹ ?P ?a
```

For the case and induction rule the cases are named `None` and `Some`.

Rules About Functions

For the functions described in Section 5.5.3 most properties are specified by equations such as

```
elem_set: (?x ∈ set_option ?xo) = (?xo = Some ?x)
map_option_case:
  map_option ?f ?y = (case ?y of None ⇒ None | Some x ⇒ Some (?f x))
Option.is_none_def: Option.is_none ?x = (?x = None)
rel_option_unfold:
  rel_option ?R ?x ?y =
  (Option.is_none ?x = Option.is_none ?y ∧
   (¬ Option.is_none ?x ⟶
    ¬ Option.is_none ?y ⟶ ?R (the ?x) (the ?y)))
```

Many of these equations are members of the simpset, therefore many properties about optional values can be proved by the simplifier. Remember to configure it with `split: option.split` if `case` terms occur for type `'a option`.

5.6 Tuples

Tuples are represented by HOL as nested pairs. The type of pairs is specified equivalent to an algebraic type (see Section 4.1) of the form

datatype `('a, 'b) prod = Pair (fst: 'a) (snd: 'b)`

As described in Section 4.1, this type is equivalent to the type of pairs of values of the type parameters `'a` and `'b`. It is also called the "product type" of the types `'a` and `'b`.

HOL supports an alternative syntax for instances of type `('a, 'b) prod`. The type instance (t_1, t_2) `prod` where t_1 and t_2 are arbitrary types may be denoted by the type expression $t_1 \times t_2$ or $t_1 * t_2$ (see Section 3.6).

A tuple with more than two components is represented in HOL by a pair where the second component is again a pair or tuple. Hence the type of 4-tuples with component types t_1, t_2, t_3, t_4 can be denoted by the type expression $t_1 \times (t_2 \times (t_3 \times t_4))$. Since $\times$ is right associative the parentheses may be omitted as in the equivalent type expression $t_1 \times t_2 \times t_3 \times t_4$.

As an example the type `(bool × nat × nat) option` is the type of optional triples of one boolean value and two natural numbers.

Note that the type `unit` (see Section 5.2) can be used to represent a "0-tuple", as the alternative form `()` of its single value suggests.

5.6.1 Values

All values of type `('a, 'b) prod` are denoted using the single constructor `Pair`. HOL supports an alternative syntax for these constructor application terms: the term `Pair` $term_1$ $term_2$ may also be specified in the form ($term_1$, $term_2$) (see Section 3.6).

Iterating this syntax a 4-tuple value may be specified as ($term_1$, ($term_2$, ($term_3$, $term_3$))). Since the (...,...) is also right associative the inner parentheses may be omitted resulting in the equivalent term ($term_1$, $term_2$, $term_3$, $term_3$).

Thus an example value of type `(bool × nat × nat) option` can be specified by the term `Some (True, 5, 42)`.

5.6.2 Destructors

The selectors `fst` and `snd` can be used to access the first or second component in a pair, respectively.

Note that in a tuple of more than two components `fst` still selects the first component, whereas `snd` selects the nested tuple with all other components. To access the third component in a 4-tuple `x` the selectors must be combined accordingly: `fst (snd (snd x))`. The third component in a triple `x` must be accessed by `snd (snd (snd x))`. Therefore there is no general way of accessing the `i`th component in an arbitrary tuple. An "easier" way for it is to use a `case` term.

A `case` term for type $t_1 \times t_2$ (which is equivalent to (t_1, t_2) `prod`) has the form

```
case term of Pair x₁ x₂ ⇒ term₁
```

where `term` is a term of type $t_1 \times t_2$ and x_1, x_2 are variables of type t_1 and t_2, respectively, which may occur in $term_1$. Using the alternative syntax for pair values the `case` term may also be specified as

```
case term of (x₁, x₂) ⇒ term₁
```

Such `case` terms can be used to "take apart" a `term` for a pair where the components are not explicitly specified, such as in the application of a function which returns a pair. The components are bound to the variables x_1, x_2 and can be used separately in $term_1$.

As usual, this also works for tuples with more than two components. The general form of a `case` term for an n-tuple is

```
case term of (x₁, ..., xₙ) ⇒ term₁
```

where the variables $x_1, \dots, x_n$ may occur in $term_1$.

HOL provides an alternative form for such `case` terms for tuples:

```
let (x₁, ..., xₙ) = term in term₁
```

Although this form looks like a generalized `let` term (see Section 3.5) this does not imply that `let` terms support arbitrary patterns on the left side of the `=` sign, like the `let` statement (see Section 2.2.11). As "patterns" only nested tuples of variables may be used, such `let` terms are always equivalent to a `case` term for a tuple.

The same variable tuple patterns can also be used in other kinds of terms where variables are bound such as in lambda terms (e.g., `λ(a,b) c. a+b+c`), in logic quantifiers (e.g., `∀ (a,b) c. a+b=c`), and in set comprehensions (e.g., `{(a,b). a=b*b}`). Note that the last example is equivalent to `{(a,b) | a b. a=b*b}`, only in this form an arbitrary term may be used instead of `(a,b)`.

5.6.3 Functions

As described in Section 4.1.3 a `case` term for a pair is only an alternative syntax for the function application term `case_prod (λ` x_1 x_2`.` $term_1$`) term`. The polymorphic function `case_prod` is provided as

```
case_prod :: ('a ⇒ 'b ⇒ 't) ⇒ 'a × 'b ⇒ 't ≡
  λf (x,y). f x y
```

where `'t` is the type of $term_1$ and thus of the whole `case` term.

This function is of interest on its own. It converts a binary function `f` specified in the style described in Section 2.1.3 to a function which takes the two arguments in the form of a single pair. This conversion is generally called "uncurrying" (see Section 3.6.1). Tuple patterns in variable bindings, as described in Section 5.6.2, are implemented in this way: The term `λ(a,b). term` is just an alternative syntax for `case_prod (λa b. term)`.

HOL provides the reverse conversion as the function

```
curry :: ('a × 'b ⇒ 't) ⇒ 'a ⇒ 'b ⇒ 't ≡
  λf x y. f (x,y)
```

Additionally there are the functions

```
apfst :: ('a ⇒ 'c) ⇒ 'a × 'b ⇒ 'c × 'b ≡ λf (x,y). (f x,y)
apsnd :: ('b ⇒ 'c) ⇒ 'a × 'b ⇒ 'a × 'c ≡ λf (x,y). (x,f y)
```

which apply their first argument to the first or second component of the second argument, respectively, and the function

```
prod.swap :: 'a × 'b ⇒ 'b × 'a ≡ λ(x,y). (y,x)
```

which interchanges the components of its argument.

The function

```
Product_Type.Times :: 'a set ⇒ 'b set ⇒ ('a × 'b) set ≡
  λA B. {(x, y). x∈A ∧ y∈B}
```

with operator name `(×)` constructs the cartesian product of two sets.

No orderings or lattice functions (see Section 3.2) are specified for values of type `('a, 'b) prod`.

BNF Functions

The type constructor `prod` is a bounded natural functor as described in Section 4.1.4. Values of type `('a, 'b) prod` can be viewed as containers of a single value of type `'a` and a single value of type `'b`.

The corresponding BNF functions are generated as:

```
map_prod ::
  ('a₁ ⇒ 'a₂) ⇒ ('b₁ ⇒ 'b₂) ⇒ ('a₁ × 'b₁) ⇒ ('a₂ × 'b₂) ≡
  λf g (x,y). (f x, g y)
pred_prod ::
  ('a ⇒ bool) ⇒ ('b ⇒ bool) ⇒ ('a × 'b) ⇒ bool ≡
  λf g (x,y). f x ∧ g y
rel_prod ::
  ('a₁ ⇒ 'a₂ ⇒ bool) ⇒ ('b₁ ⇒ 'b₂ ⇒ bool)
    ⇒ ('a₁ × 'b₁) ⇒ ('a₂ × 'b₂) ⇒ bool ≡
  λf g (x₁,y₁) (x₂,y₂). f x₁ x₂ ∧ g y₁ y₂
```

No set functions are generated, they are trivial, returning the corresponding singleton set.

5.6.4 Rules

Algebraic Type Rules

Since there is only one constructor there are no distinctness rules. The simplifier rules for the constructor of type *('a, 'b) prod* are the rule sets *prod.inject*, *prod.sel* and *prod.case* (see Section 4.1.5):

```
((?x1, ?x2) = (?y1, ?y2)) = (?x1 = ?y1 ∧ ?x2 = ?y2)
fst (?x1, ?x2) = ?x1
snd (?x1, ?x2) = ?x2
(case (?x1, ?x2) of (x, y) ⇒ ?f x y) = ?f ?x1 ?x2
```

The case, split, and induction rules are

```
prod.exhaust:
  (⋀x1 x2. ?y = (x1, x2) ⟹ ?P) ⟹ ?P
prod.split:
  ?P (case ?t of (x, y) ⇒ ?f x y) =
  (∀x1 x2. ?t = (x1, x2) ⟶ ?P (?f x1 x2))
prod.induct:
  (⋀x1 x2. ?P (x1, x2)) ⟹ ?P ?a
```

For the case and induction rule the single case is named *Pair*.

The case and induction rules are also provided for tuples of size 3 to 7 in the form

```
prod_cases3:
  (⋀x1 x2 x3. ?y = (x1, x2, x3) ⟹ ?P) ⟹ ?P
prod_induct3:
  (⋀x1 x2 x3. ?P (x1, x2, x3)) ⟹ ?P ?a
```

In all of them the single case is named *fields*.

Since all these case and induction rules are associated with the corresponding tuple types a proof of the form

```
proof (cases x)
case (fields x1 x2 x3)
...
show ?thesis ⟨proof⟩
qed
```

replaces the variable `x` of type $t_1 \times t_2 \times t_3$ by tuple terms `(x1, x2, x3)` and allows to reason about the components of `x`.

Rules About Functions

As described in Section 5.6.2 components of a tuple are easiest accessed by using a tuple pattern which is equivalent to applying the function `case_prod` (see Section 5.6.3). In proofs it is often necessary to convert a proposition involving `case_prod` to a proposition for the components. The simplifier rule

```
split_def: case_prod = (λc p. c (fst p) (snd p))
```

allows to directly replace occurrences of `case_prod` by rewriting. It is not part of the simpset (see Section 2.3.6), its use by the simplifier must be explicitly configured.

If tuples are not specified with the help of tuple patterns but by single variables of a tuple type the case, split, and induction rules described above can be used to convert such variables to explicit tuples where variables denote the components. Alternatively HOL provides the rule

```
split_paired_all: (⋀x. ?P x) ≡ (⋀a b. ?P (a, b))
```

Rewriting by this rule replaces all bound variables of a type $t_1 \times t_2$ by two variables for the components. Since tuples are nested pairs an iterated rewriting does the same for arbitrary tuples. Again, this rule is not part of the simpset and it should be added with care, an iterated rewriting is best done by the method `(simp only: split_paired_all)` without combining it with other simplifier rules.

Instead, the rules

```
split_paired_All: (∀x. ?P x) = (∀a b. ?P (a, b))
split_paired_Ex: (∃x. ?P x) = (∃a b. ?P (a, b))
```

are in the simpset and are thus automatically applied by the simplifier to replace variables of tuple type bound by logic quantifiers (see Section 5.1.3). If that is not intended these rules must be deactivated in the form `(simp del: split_paired_Ex)`.

5.7 Functions

The type *('a, 'b) fun* with alternative syntax *'a ⇒ 'b* (see Section 2.1.3) is not specific for HOL, it is provided by Isabelle at the basis of the whole Isabelle type system. It is polymorphic and denotes the type of functions with argument type *'a* and result type *'b*.

However, HOL complements this basic support for functions by many functions and rules for the type *('a, 'b) fun*.

Note that functions introduced by type definitions such as constructors and destructors of algebraic types (see Section 4.1) or the morphisms of a subtype (see Section 4.3) are normal functions of type *('a, 'b) fun* and all mechanisms described here for functions apply to them as well.

The type *('a, 'b) fun* is not equivalent to an algebraic type. Since functions are always total, the arguments of a constructor function would have to specify function values for all arguments of type *'a* which may be infinitely many.

5.7.1 Values

The values of type *'a ⇒ 'b* are denoted by lambda terms (see Section 2.1.3). HOL introduces the name *id* for the polymorphic identity function *λx. x*.

5.7.2 Functions on Functions

The functions *image*, *vimage*, and *range* have already been described in Section 5.4.3, they can be viewed to “lift” functions on values to functions on value sets.

Functions of arbitrary types can be composed by the polymorphic function

```
comp :: ('b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c
  ≡ λf g x. f (g x)
```

if the “intermediate” type *'b* matches. The operator name for infix notation is *(∘)*.

There is also the variant *fcomp* with reversed arguments where the left argument is applied first and the function

```
scomp :: ('a ⇒ 'b × 'c) ⇒ ('b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'd
  ≡ λf g x. (case_prod g) (f x))
```

which composes a binary function with a function with pairs as values, using the uncurry function *case_prod* to convert the binary function to a function with argument pairs (see Section 5.6.3). The operator names *(∘>)* for *fcomp* and *(∘→)* for *scomp* are only available after using the command

unbundle *state_combinator_syntax*

on theory level.

Finite iteration of a function of type `'a ⇒ 'a` can be specified by the polymorphic function

```
funpow :: nat ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
```

with operator name `(^^)` (with reversed arguments) for infix notation. Thus `funpow 3 f = f ^^ 3 = f ∘ f ∘ f`.

Injectivity and Surjectivity

For the basic function properties of injectivity, surjectivity, and bijectivity HOL provides the predicates

```
inj :: ('a ⇒ 'b) ⇒ bool ≡ λf. ∀x y. f x = f y ⟶ x = y
surj :: ('a ⇒ 'b) ⇒ bool ≡ λf. range f = UNIV
bij :: ('a ⇒ 'b) ⇒ bool ≡ λf. inj f ∧ surj f
```

and also the domain-restricted forms

```
inj_on :: ('a ⇒ 'b) ⇒ 'a set ⇒ bool
  ≡ λf A. ∀x∈A. ∀y∈A. f x = f y ⟶ x = y
bij_betw :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool
  ≡ λf A B. inj_on f A ∧ image f A = B
```

Injective functions can be inverted on their range. HOL provides the more general inversion function

```
the_inv :: ('a ⇒ 'b) ⇒ ('b ⇒ 'a)
  ≡ λf y. THE x. f x = y
```

which returns the argument mapped by function `f` to the value `y`. Note the use of the definite description operator `THE` (see Section 3.3.2). If there is no such argument (because `y` is not in the range of `f`) or if it is not unique (because `f` is not injective), it causes the function to be underspecified. Thus the partial application `(the_inv f)` is the inverse of an arbitrary function `f`. It is total but only specified for values of type `'b` where the inversion exists and is unique. It is only fully specified if `bij f`.

Additionally there is the domain-restricted form

```
the_inv_into :: 'a set ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a)
  ≡ λA f y. THE x. x ∈ A ∧ f x = y
```

where the partial application `(the_inv_into A f)` is the inverse of `f` restricted to arguments in set `A`. It is only fully specified if `image f A = UNIV` and `inj_on f A`.

Function Updates

HOL provides the function

```
fun_upd :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b)
  ≡ λf a b x. if x = a then b else f x
```

which returns a function where the value of a single argument `a` of `f` has been changed to `b`. Note that this "function update" does not "change" the function `f`, it returns a new function which differs from `f` only for the argument `a`.
HOL provides the alternative syntax

`f(terma`$_1$` := termb`$_1$`, ..., terma`$_n$` := termb`$_n$`)`

for the term `fun_upd ... (fun_upd f terma`$_1$` termb`$_1$`) ... terma`$_n$` termb`$_n$. The changes are applied from left to right, i.e. `f(x := y, x := z) = f(x := z)`.
HOL also provides an update on a set of arguments, where the new values are specified by another function:

```
override_on :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ 'a ⇒ 'b
  ≡ λf g A x. if x ∈ A then g x else f x
```

The Type Constructor `fun` as a Functor

The type constructor `fun` is no bounded natural functor. Values of type `('p`$_1$`, 'p`$_2$`) fun` can be viewed as containers for the function values of type `'p`$_2$. Since there may be a separate function value for every argument value of type `'p`$_1$ the set of contained values may be arbitrary large depending on the type `'p`$_1$, it is not bounded.
However, `fun` is still a functor and has the map function

`map_fun :: ('q`$_1$` ⇒ 'p`$_1$`) ⇒ ('p`$_2$` ⇒ 'q`$_2$`) ⇒ ('p`$_1$` ⇒ 'p`$_2$`) ⇒ 'q`$_1$` ⇒ 'q`$_2$
`  ≡ λf`$_1$` f`$_2$` f = f`$_1$` ∘ f ∘ f`$_2$

which lifts two functions `f`$_1$`, f`$_2$ to a function `(map_fun f`$_1$` f`$_2$`)` on functions and which satisfies the laws for a functor:

`map_fun.id: map_fun id id = id`
`map_fun.comp:`
`  map_fun ?f ?g ∘ map_fun ?h ?i = map_fun (?h ∘ ?f) (?g ∘ ?i)`

Note the different treatment of the first type parameter `'p`$_1$ by reversing the direction of the application of function `f`$_1$.
Additionally there are functions which lift predicates and relations (see Section 3.1) in a similar way as the predicators and relators described for algebraic types in Section 4.1.4. These are

```
pred_fun :: ('p₁ ⇒ bool) ⇒ ('p₂ ⇒ bool) ⇒ ('p₁ ⇒ 'p₂) ⇒ bool
  ≡ λp₁ p₂ f. ∀x. p₁ x ⟶ p₂ (f x)
```

and

```
rel_fun :: ('p₁ ⇒ 'q₁ ⇒ bool) ⇒ ('p₂ ⇒ 'q₂ ⇒ bool)
  ⇒ ('p₁ ⇒ 'p₂) ⇒ ('q₁ ⇒ 'q₂) ⇒ bool
  ≡ λr₁ r₂ f g. ∀x y. r₁ x y ⟶ r₂ (f x) (g y)
```

Note again the different treatment of the first type parameter *'p₁* by combining it with the second by implication ⟶ instead of conjunction.

As an example using the predicate *evn* from Section 3.7.1, the partial application *pred_fun evn evn* is the predicate of type *(nat ⇒ nat) ⇒ bool* which tests whether a function on natural numbers maps all even numbers to even numbers.

BNF Functions

Although the type constructor *fun* is no bounded natural functor, it becomes one if the first type parameter is fixed, such as for the type *(nat, 'p₂) fun* or *nat ⇒ 'p₂*. It has only one type parameter for the function values, the type of the function arguments is always the same. Now for a function *f* of this type the set of contained values is simply its range *range f* and that is bounded by the cardinality of the argument type.

This situation is described by saying that *fun* is a bounded natural functor with one "dead" type parameter (the first one). The second type parameter is called "live". In general a bounded natural functor may have several dead and live type parameters. A set function only exists for each live type parameter and the map function and the predicator and relator lift only functions for the live type parameters.

Therefore *fun* has the set function (see Section 5.4.3)

```
range :: ('p₁ ⇒ 'p₂) ⇒ 'p₂ set ≡ λf. {y. ∃x. y = f x}
```

and the map function, predicator and relator can be constructed by the partial applications

```
(map_fun id) = (λf₂ f x. f₂ (f x))
(pred_fun (λ_. True)) = (λp₂ f. ∀x. p₂ (f x))
(rel_fun (=)) = (λr₂ f g. ∀x. r₂ (f x) (g x))
```

The map function is equivalent to the composition *(∘)*. For the predicator and relator HOL does not define separate names. Note that they are also polymorphic for the argument type, so they can be used for functions of arbitrary types *'p₁ ⇒ 'p₂*. They lift a predicate or relation by applying it to all function values.

Since functions with multiple arguments in curried form (see Section 3.6.1) have functions as intermediate result values the lifting can be iterated over multiple arguments. For example, a relation `r` on the result type `t` can be lifted to binary functions of type $t_1 \Rightarrow t_2 \Rightarrow t$ by `rel_fun (=) (rel_fun (=) r)` which is equivalent to the relation on functions $\lambda f\ g.\ \forall x\ y.\ r\ (f\ x\ y)\ (g\ x\ y))$.

Functions for Orderings and Lattices

The ordering relations (see Section 3.2.2) are defined for functions by lifting the ordering relations for the function values. Thus the ordering `(<)` on functions is equivalent to `rel_fun (=) (<)` and analogously for $(\le)$, `(>)`, $(\ge)$. In other words, $f < g$ holds if $\forall x.\ (f\ x) < (g\ x)$. Note that even if `(<)` is a total ordering on the function values, the lifted ordering is partial, because for some arguments the function values may be less and for other arguments not.

In a similar way the lattice operations $(\sqcap)$, $(\sqcup)$, $(\bigsqcap)$, and $(\bigsqcup)$ (see Section 3.2.3) are lifted from the function values to functions:

$f \sqcap g \equiv \lambda x.\ f\ x \sqcap g\ x$
$f \sqcup g \equiv \lambda x.\ f\ x \sqcup g\ x$
$\bigsqcap A \equiv \lambda x.\ \bigsqcap f{\in}A.\ f\ x$
$\bigsqcup A \equiv \lambda x.\ \bigsqcup f{\in}A.\ f\ x$

Since `(<)` is not a total ordering on functions the functions `min` and `max` are *not* equivalent to $(\sqcap)$ and $(\sqcup)$ and the functions `Min` and `Max` are not available for sets of functions.

Since orderings and lattice operations are defined for type `bool` (see Section 5.1.3 so that $(\le)$ is equivalent to the implication $(\longrightarrow)$, the lifting works specifically for predicates and relations (see Section 3.1). Predicates are ordered by the "stronger as" relation where $(p_1 \le p_2) \longleftrightarrow (\forall x.\ p_1\ x \longrightarrow p_2\ x)$ and the lattice operations correspond to point-wise conjunction or disjunction, such as $(p_1 \sqcap p_2) = (\lambda x.\ p_1\ x \wedge p_2\ x)$. As described above for functions with multiple arguments the boolean operations can be lifted in the same way to binary or n-ary relations. Binary relations are ordered by the "stronger as" relation where $(r_1 \le r_2) \longleftrightarrow (\forall x\ y.\ r_1\ x\ y \longrightarrow r_2\ x\ y)$ and the lattice operations are as above, such as $(r_1 \sqcap r_2) = (\lambda x\ y.\ r_1\ x\ y \wedge r_2\ x\ y)$.

Monotonicity

Another way to derive predicates on functions is by applying `rel_fun` to compare a function to itself. HOL supports this by the function

$monotone :: ('p_1 \Rightarrow 'p_1 \Rightarrow bool) \Rightarrow ('p_2 \Rightarrow 'p_2 \Rightarrow bool)$
$\quad \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow bool$
$\quad \equiv \lambda r_1\ r_2\ f.\ rel\_fun\ r_1\ r_2\ f\ f$

The partial application *monotone* r_1 r_2 is the predicate which tests a function whether arguments related by r_1 produce results related by r_2:

$monotone\ r_1\ r_2 = (\lambda f.\ \forall x\ y.\ r_1\ x\ y \longrightarrow r_2\ (f\ x)\ (f\ y))$

The usual notion of monotonicity results when *monotone* is applied to the ordering relations. HOL defines the specific predicates on functions

$mono :: ('p_1 \Rightarrow 'p_2) \Rightarrow bool \equiv monotone\ (\leq)\ (\leq)$
$strict\_mono :: ('p_1 \Rightarrow 'p_2) \Rightarrow bool \equiv monotone\ (<)\ (<)$
$antimono :: ('p_1 \Rightarrow 'p_2) \Rightarrow bool \equiv monotone\ (\leq)\ (\geq)$

HOL also provides the domain-restricted variants:

$monotone\_on :: 'p_1\ set \Rightarrow ('p_1 \Rightarrow 'p_1 \Rightarrow bool) \Rightarrow ('p_2 \Rightarrow 'p_2 \Rightarrow bool)$
$\quad \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow bool$
$mono\_on :: 'p_1\ set \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow bool$
$strict\_mono\_on :: 'p_1\ set \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow bool$

5.7.3 Rules

Since functions are the basic type of Isabelle there are several rules which are already built in and need not be provided by HOL. Some of them are implicitly applied to propositions without the need of using proof methods, such as the rewriting rule

$(\lambda x.\ ?P\ x)\ ?y = ?P\ ?y$

Many others are known by the automatic proof methods (see Section 2.3.7) so that properties relating to functions can usually be proved by them.

The most basic rules provided by HOL about functions are

$ext\text{:}\ (\bigwedge x.\ ?f\ x = ?g\ x) \Longrightarrow ?f = ?g$
$fun\_cong\text{:}\ ?f = ?g \Longrightarrow ?f\ ?x = ?g\ ?x$
$arg\_cong\text{:}\ ?x = ?y \Longrightarrow ?f\ ?x = ?f\ ?y$
$cong\text{:}\ [\![?f = ?g;\ ?x = ?y]\!] \Longrightarrow ?f\ ?x = ?g\ ?y$

Rules About Functions on Functions

Reasoning about the functions described in Section 5.7.2 can often be done by substituting their definition and then using automatic proof methods. Alternatively HOL provides many rules for direct reasoning about these functions, such as

```
comp_assoc: ?f ∘ ?g ∘ ?h = ?f ∘ (?g ∘ ?h)
image_comp: ?f ` ?g ` ?r = (?f ∘ ?g) ` ?r
```

and in particular introduction rules (see Section 2.3.3) such as

```
injI: (⋀x y. ?f x = ?f y ⟹ x = y) ⟹ inj ?f
surjI: (⋀x. ?g (?f x) = x) ⟹ surj ?g
bijI: ⟦inj ?f; surj ?f⟧ ⟹ bij ?f
monoI: (⋀x y. x ≤ y ⟹ ?f x ≤ ?f y) ⟹ mono ?f
```

the destruction (see Section 2.3.4) and elimination (see Section 2.4.4) rules

```
comp_eq_dest: ?a ∘ ?b = ?c ∘ ?d ⟹ ?a (?b ?v) = ?c (?d ?v)
injD: ⟦inj ?f; ?f ?x = ?f ?y⟧ ⟹ ?x = ?y
monoD: ⟦mono ?f; ?x ≤ ?y⟧ ⟹ ?f ?x ≤ ?f ?y
comp_eq_elim:
  ⟦?a ∘ ?b = ?c ∘ ?d; (⋀v. ?a (?b v) = ?c (?d v)) ⟹ ?R⟧ ⟹ ?R
surjE: ⟦surj ?f; ⋀x. ?y = ?f x ⟹ ?C⟧ ⟹ ?C
monoE: ⟦mono ?f; ?x ≤ ?y; ?f ?x ≤ ?f ?y ⟹ ?thesis⟧ ⟹ ?thesis
```

and the simplifier rules (see Section 2.3.6)

```
comp_apply: (?f ∘ ?g) ?x = ?f (?g ?x)
fun_upd_apply:
  (?f(?x := ?y)) ?z = (if ?z = ?x then ?y else ?f ?z)
bij_id: bij id
override_on_apply_in:
  ?a ∈ ?A ⟹ override_on ?f ?g ?A ?a = ?g ?a
```

5.8 Relations

todo

5.9 The Sum Type

todo